1984

Computer systems performance measurement: theory and practice

P. J. McKerrow
University of Wollongong, phillip@uow.edu.au

COMPUTER SYSTEMS PERFORMANCE MEASUREMENT:

THEORY AND PRACTICE

A thesis submitted in fulfillment of the
requirements for the award of the degree of

DOCTOR OF PHILOSOPHY

from

THE UNIVERSITY OF WOLLONGONG

by

PHILLIP JOHN McKERROW, B.E. HONOURS CLASS II DIVISION 1, M.E.

COMPUTING SCIENCE

1984

In Memory of

Ian Paul and John Mark

born 31.8.81, died 1.9.81

ACKNOWLEDGEMENTS

I wish to thank my Supervisor, Professor Juris Reinfelds, for his encouragement and direction during this research. He had the uncanny ability to question the current state of the work in a way which forced me to clarify my ideas, prompting further avenues of thought and deeper insight.

The logic-state analyser used in the research was bought with a grant from the Department of Science and Technology, Australian Research Grants Committee. The Apple microcomputer, and other facilities were provided by the Department of Computing Science.

During the course of this research I have regularly prayed about problems and meditated upon insights. I have always found God to be one step ahead of me, and ready to give insight and understanding.

Finally I would like to thank my wife and family for the hours of my time they have given up so that I could write this tome. No acknowledgement is complete without heartfelt thanks to the typist: Mrs. Lynn Maxwell, who can type faster than I can think; and to the graphic artist, Mr. John Murray.

This thesis has been typeset on a Compugraphic phototypesetter using the troff word processor.

LIST OF CONTENTS

3.5. Monitoring Program Execution - Basic Concepts - Plattner and Nievergelt (1981)

3.6. A Sequential Program Model - Franta et al (1982)

3.7. A Measure of Computational Work - Hellerman (1972)

3.8. Program Behaviour: Models and Measurement - Spirn (1977)

4. MEASUREMENT TOOLS AND TECHNIQUES

4.1. Measurement Tool Modules

4.2. Measurement Tool Characteristics

4.3. Hardware Tools and Techniques

4.3.1. Characteristics of Hardware Tools

4.3.2. Some Actual Hardware Tools

4.4. Software Tools and Techniques

4.4.1. Characteristics of Software Tools

4.4.2. Some Actual Software Tools

4.5. Hybrid Tools and Techniques

13. APPENDICES

13.1. Measurement Software

13.2. Communications Test Software

ABSTRACT

The results of the last three decades of research into computer-systems-performance measurement are combined into a unified body of knowledge: theory and practice. Unification is based upon a formulation of performance measurement.

A computer system is an implementation of a mathematical object, which can be measured. Performance measurement entails ascertaining the extent of this object during execution. An object (computer system, task, program, procedure) is defined recursively in terms of lower level objects. A set of measures, which apply at every level of the hierarchy, has been defined mathematically. This set of equations is a formulation of performance measurement. A number of graphical representations of these equations, for use in evaluating the measured data, are demonstrated.

The formulation provides a general, overall context within which measurement and evaluation can take place. The purpose of measurement is not to collect numbers, but to gain insight into the actions of the object under study. The recording of appropriate stimulus information, and the use of graphical techniques to analyse the data, gives meaning to the actions of the object.

The formulation has been validated in a number of ways:

- measurement experiments have been conducted,
- measures proposed by the formulation have been compared to current measurement practice,
- other formulations are compared to it, and
- corollaries have been hypothesised and tested.

Results of these validation procedures indicate a high degree of correlation between the formulation and current practice. A hybrid performance-analyser, designed on the basis of the formulation, has proved to be a powerful measurement tool for use in performance evaluation, in sys-

tem optimisation, and in program execution monitoring; for debugging software; and for finding software related hardware faults. A number of future research areas, which flow out of the formulation, have been proposed.

The thesis commences with an introduction to the field of performance measurement and an overview of various aspects of it. Then, the formulation of performance measurement is described in detail. Other formulations proposed by researchers in the performance-evaluation field are discussed, and the underlying conceptual models of program execution are compared. Following this is a review of measurement tools and techniques. Next comes the design of a hybrid performance-analyser, which is built around a logic-state analyser, and is based on a philosophy of hybridisation derived from the formulation of performance measurement. Then, the design of computer systems for performance measurement is discussed.

Case studies are included to illustrate performance-measurement methods, and software-debugging techniques. These case studies demonstrate the practicality, and power, of a performance-measurement methodology based on the formulation of performance measurement. Finally, the formulation is extended to cover parallel processors, and measurement in a number of other applications.

PUBLICATIONS FROM THE THESIS

- McKerrow P.J. (1983), Evaluation of Interrupt-Handling Routines with a Logic-State Analyser, Performance Evaluation, North-Holland, Vol 3, pp 277-288.

- McKerrow P.J. (1984), Monitoring Program-Execution with a Hybrid Measurement-Tool, Australian Computer Science Communications, Vol 6. No. 1, pp 22.1-10.

{Modified versions of these papers are included as Chapters 5 and 7 respectively.}

1. Introduction

1.1. Performance Measurement

Computer systems performance evaluation has become a very large field, which can be divided into four broad areas: measurement of system parameters, evaluation of the data, modelling system behaviour, and modifications to improve performance. In this dissertation, I have chosen to concentrate on the measurement of system parameters. In the early days of computing, the main goal was to get a working program with little thought about the efficiency of the program, however, there were some exceptions. Von Neumann (1946) compared the speed with which a number of early computers, including ENIAC, performed multiplications when computing ballistic trajectories. Herbst et al (1955) measured the instruction mix of programs running on the Maniac computer.

In the early sixties performance measurement of systems was commenced in earnest. As computing systems became more readily available to users, ways were sought to increase the productivity of both the computer and the programmer, and hence, to reduce the cost of computing by more efficient use of resources. Computer throughput was increased by the development of operating systems to handle resource sharing: initially simple batch systems, and more recently, time sharing and multiprogramming. Program development time has been shortened through the use of high-level languages, structured programming, and the emergence of software engineering as a discipline.

Concurrent with these developments, and spurred on by the high cost of computing, has been a desire to evaluate how well the systems have been performing, and to find ways of improving that performance. During the sixties performance measurement studies were carried out on many installations. By 1967 the field had grown to the point where Calingert (1967) was able to publish a survey of the then common techniques, and a few years later Miller (1972) published a bibliography of over two hundred and fifty papers in the field. The early seventies

saw a hive of measurement activity which diminished to a mere trickle of papers by the mid seventies as researchers turned to modelling techniques.

Measurement is a fundamental technique in any science (Curtis 1980). The fact that little work has been reported on the measurement of computer systems in the last few years has been seen by some as an indication that all the work that needs to be done has been done. This is not true - computer performance measurement is still just a collection of techniques with no unified body of knowledge. The goal of this dissertation is to codify the field into a unified body of knowledge, both theoretical and practical. Research effort dwindled, not because all the problems were solved, but because of a number of other factors.

- Measurement ideas were several years ahead of the available technology. It is interesting to note, in some papers from the heyday of measurement, the gradual transition from what we have done, to what we are doing, to what we think we might be able to do when we finish developing the tool. Consequently, most of the ideas are not new, but the technology of the early seventies was not cheap enough for the development of powerful, general-purpose tools.

- The complexity of computer systems increased rapidly, making measurement more difficult.

- Researchers were attracted by the mathematical tractability of modelling techniques, particularly analytical queueing models. This field provided a rich source of research at a time when measurement was being frustrated by the increasing complexity of computer systems. The lack of tools powerful enough to handle the complexity of the systems simply made measurement too hard.

- The literature of the time consisted of descriptions of measurement techniques and their results. No unified body of knowledge had been established and no theoretical basis for measurement had been developed. Hence, there was no framework within which to tackle the measurement problems posed by the new, more complex systems.

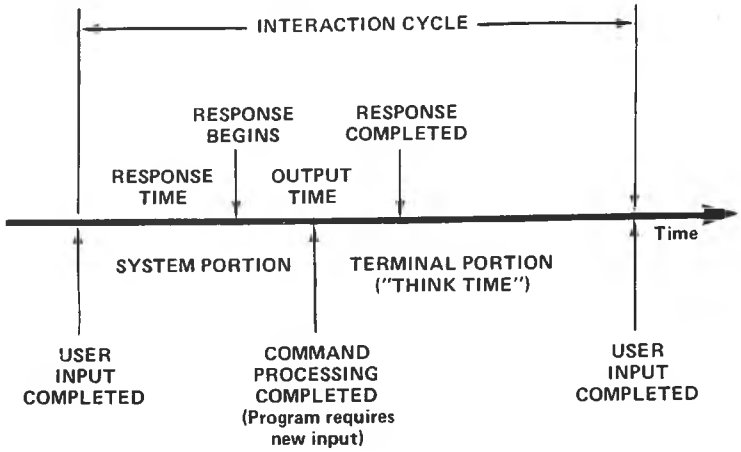


Figure 1.1 User-Computer Interaction Cycle (Ferrari 1978)

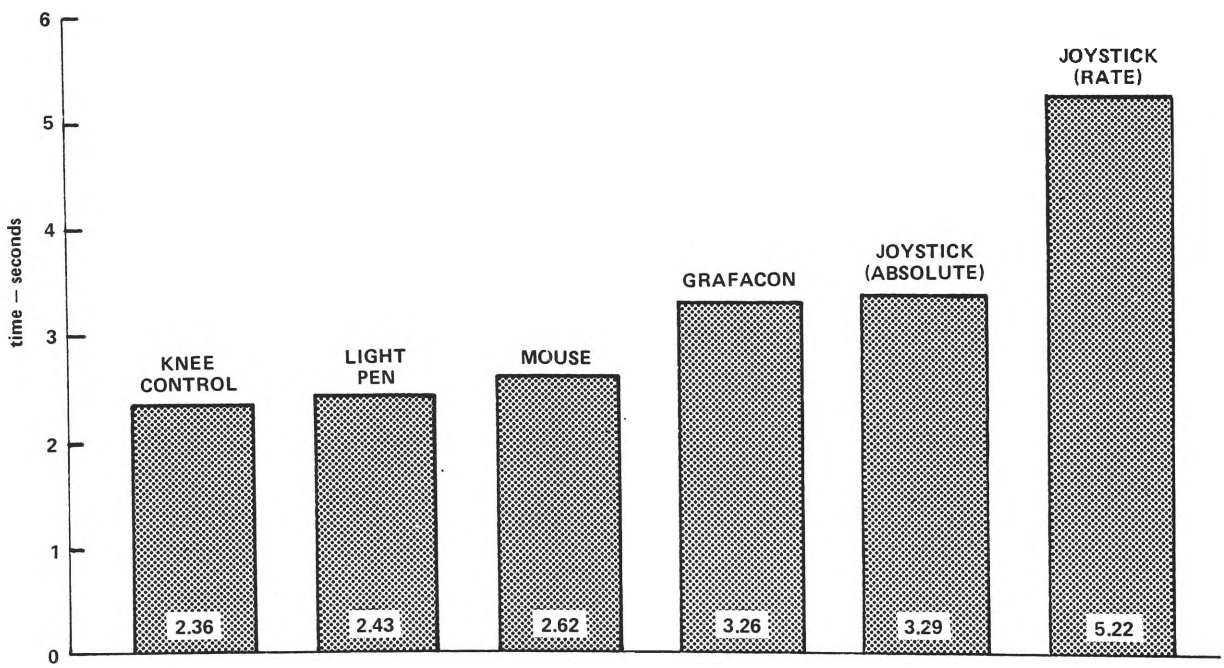


Figure 1.2 Comparison of Average Times Taken by Inexperienced Users to Locate a Cursor at a Character using a variety of graphics input devices (English et al 1967)

During the last decade, advances in technology have made computing power so cheap that all new test instruments include microprocessors as-powerful-as the mainframe computers of the late sixties. One instrument developed in the last decade, the logic-state analyser, is now more powerful than any of the hardware measurement tools of a decade ago. These two advances in technology mean that technology is no longer a limitation in measurement. The explosion in the use of microcomputers increases dramatically the need for effective performance measurement. However, the design of tools suitable for measuring computer performance must be grounded in a unified formulation of measurement if lasting results are to be achieved. Such a formulation is developed in the next chapter. Current measurement techniques are evaluated in the light of this formulation in subsequent chapters, and some of the implications of the formulation for future measurement techniques and tools are investigated. The result is a unified body of performance measurement knowledge (theoretical and practical).

1.2. Measurement Categories

The first step in the development of a unified formulation of measurement is to gather all the currently independent measurement categories together under one umbrella. Then common principles can be extracted. We need to stop discussing the various measurement situations as completely separate problems and see that the differences are differences in the application of theory and tools not conceptual differences in either theory or tools. In the following paragraphs the major applications of performance measurement are briefly discussed. Many of these areas overlap, and hence the discussion is aimed at showing the breadth of measurement applications rather than at a definitive classification.

Human engineering of computer systems involves the measurement of the interaction between the user and the system. The user influences the performance of the system by producing inputs: requests for program execution, data, system commands, new programs etc. The system's response to these inputs is important, particularly on an interactive system (figure 1.1) where the user expects fast response to commands which are input at highly irregular intervals.

If the response is too slow, the user gets frustrated and will eventually seek another system. Lack of feedback to the user may result in the user executing more commands, to check that the previous one worked, significantly increasing the workload. The classic example of this occurred in 1963 when a major American airline was trying unsuccessfully to go on line nation-wide with its new computerised reservation system (Warner 1974). Everything went well until they tried to bring the last and busiest region, New York City, on line. The system crashed, hopelessly overloaded. Measurement revealed that just before New York was brought in the existing load was 90%, not 40% as predicted by simulation. Each operator, after keying in a reservation, would immediately make an enquiry to see if the system really did have the data. The solution: a wiggle of the ball on the typewriter let the operator know the data was in.

Ergonomic Design of input devices, and output displays, involves the measurement of human and system response times in an attempt to evaluate a vague idea called ease of use (figure 1.2).

Selection of new computing equipment for a company, often involves the running of bench-marks on comparative systems in an effort to measure workload characteristics such as: capa-city, throughput, batch turn around time, number of interactive users, response time of high usage programs, etc. Bench marks range from the execution of a program typical of the appli-cation, for example a floating point number cruncher in a scientific application, to complex job control scripts, for example the reproduction of a complete day's workload of an existing sys-tem.

Capacity planning includes the measurement of the usage-by-the-system of the available capa-city, and the use of that data for managing and planning workload growth. Work load, the total of resource demands by users, changes unpredictably over short periods, so characterisa-tion of workload is done over long periods. Workload parameters (Svobodova 1976a table 2.1) include: Job CPU Time, Job I/O requests, CPU service time, I/O service time, job priority, job memory usage, job paused time, etc. This information is used during system generation to establish the size of tables, and to determine the configuration of the system. It is also used

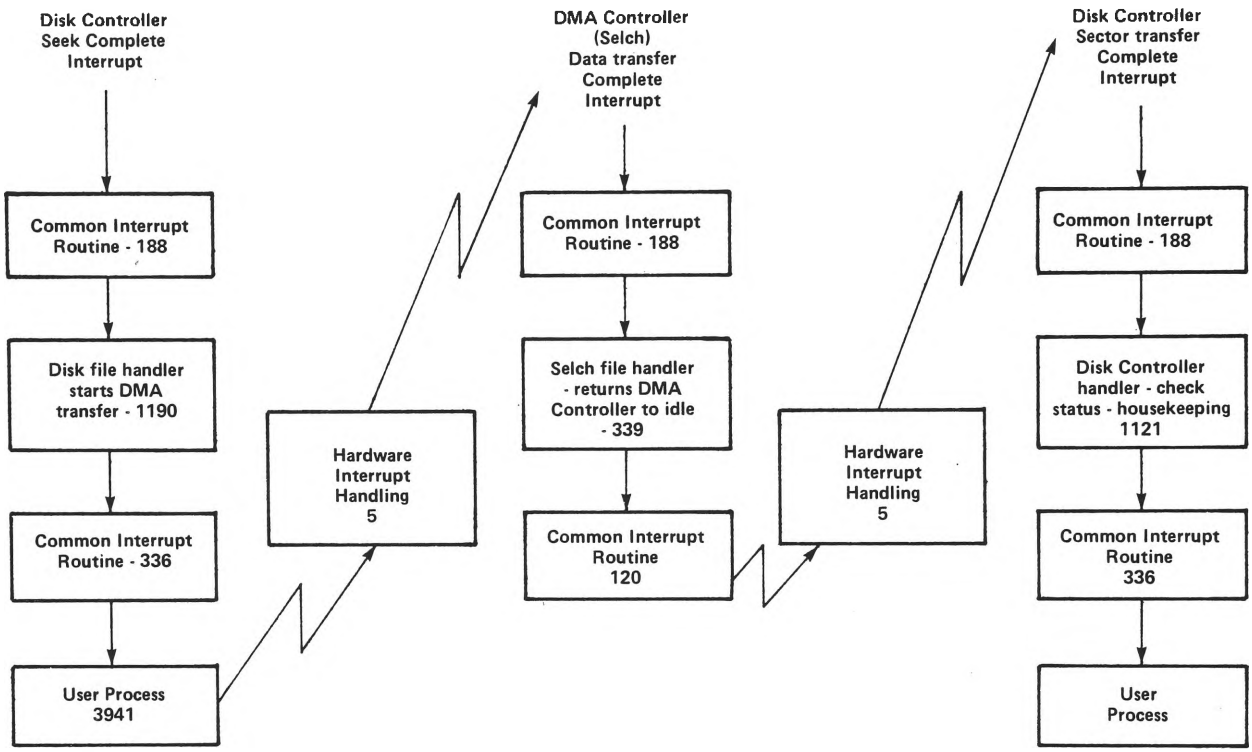


Figure 1.3 Interrupt Handling Sequence during data transfer from a disk under DMA control, measured on the UNIX * operating system running on a Perkin-Elmer 7/32 (all times in microseconds) * UNIX is a trademark of Bell Laboratories

during production to obtain the best system performance by controlling priority levels, resource allocation strategies etc. An important part of the management of a system is billing the customer for resources used. System accounting involves the measurement of resources by individual jobs and the generation of accounts and statistics.

Operating system performance measurement entails the examination of a running system to find bottlenecks or 'performance bugs'. Measurements range from determining interrupt handling time of an I/O device (figure 1.3) to measuring system overhead during the execution of a controlled workload. When a system is suspected to perform badly under certain conditions, these conditions can be simulated and measurements made in an attempt to pinpoint the cause of the poor performance.

Methods of improving the performance, for example a different algorithm, can then be hypothesised, implemented, measured, and compared to the original measurements. System debugging, tuning, and optimisation activities can greatly improve the performance of a system, and as a consequence, may delay an upgrade for a period of time, saving the company money.

One of the major tasks carried out on any computer system is the **development of new programs**. Dynamic analysis of executing programs (program executions monitoring) is used to find faults, to pinpoint time consuming routines, and to compare the performance of algorithms.

System designers study how existing systems are used, and how they perform, in order to design better systems (Bisiani et al 1983). One of the goals of performance measurement research is for the inclusion of performance instrumentation to be seen as a basic design requirement rather than as an afterthought. Some researchers (Boulaye et al 1977, Geck 1979) have suggested that the ultimate goal of performance evaluation research is to be able to measure and adaptively control performance indices in real time, in order to maintain optimum performance under varying system load conditions. Instruction mix studies are used in the design of new processor architectures, and in the development of new programming languages.

During the design of a new system, **models** are often used to predict performance. Data measured on existing systems is used to build these models, and data collected from the system, once it is built, is used to verify and enhance the models. Once the model has been validated, it can then be used in capacity planning, and in operating system performance studies.

Another important area of measurement, particularly with microcomputer systems, is the determination of **software/hardware compromises.** Microcomputer development systems include monitoring tools, both hardware (logic state analysers) and software (in circuit emulators).

Svobodova (1976b) has divided computer system measurement needs into three categories:

- **Diagnostic measurement,** necessary to secure correct operation while a computer system is in use and to restart correct operation in case of system failure.
- **Performance measurement,** necessary to ensure efficient operation and fast response of a computer system under dynamically changing demands.
- **Analytic measurement,** necessary to develop understanding of the processing requirements and their impact on system behaviour and performance.

In all these categories performance is defined differently - usually in terms of the perceived goals of the measurement evaluation study. However, the question - Where does the time go? - is common to all, as is the more fundamental question - Why did it go that way? Part of the verification of the formulation of performance measurement, developed in the next chapter, is to study measurements made in each of these categories, as reported by other researchers, to see if they fit into the framework defined by the formulation.

1.3. Measurement Tools and Techniques

One of the outcomes of a formulation of measurement, in fact a very desirable outcome, should be the development of measurement tools and techniques, based upon that formulation, which can be used to make measurements in the applications described in section 1.2.

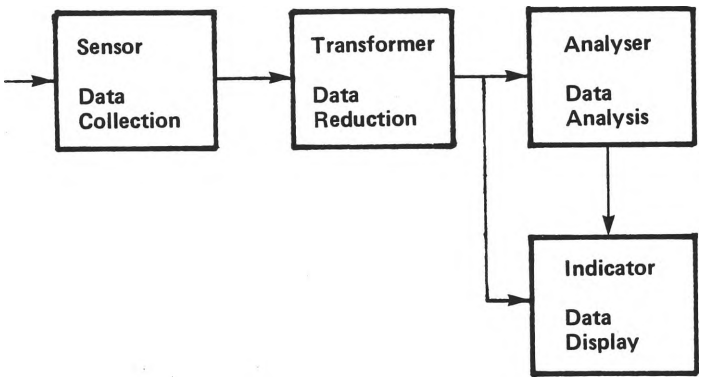


Figure 1.4 Four Conceptual Parts of a Measuring Instrument

In this section, measurement tools, techniques, and problems are introduced. They are discussed in the light of the performance measurement formulation in chapters 4 to 7.

The main sources of measurement data are existing accounting software, hardware monitors attached to the system, and software monitors. Normally, accounting data does not contain sufficient information, and thus, a hardware or software monitor is required.

Conceptually, a measuring instrument consists of four parts (figure 1.4). The sensor's task is to sense the magnitude of the quantity being measured, and thus, it includes the software or hardware probes. In the transformer section, the measured data is manipulated and reduced to the desired form, e.g. a state analyser's ability to distinguish desired states from a continuous data stream. After data reduction, the measured information can be displayed directly or it can be analysed and the results of the analysis displayed.

1.3.1. Hardware Monitors

Hardware tools consist of additional hardware attached to the computer's backplane via test probes. Attaching the probes requires a detailed knowledge of the backplane and a clear specification of which signals are to be monitored. Consequently hardware probes tend to make maintenance people nervous. Hardware monitors can detect events occurring at microscopic levels with high accuracy, e.g. individual instruction fetches. Thus, potentially their scope is broad since most of the interesting points in the system can usually be reached. The effect of the increasing scale of chip integration can be offset if probe points are included at design time.

Hardware monitors are often used to check the accuracy of software monitors. Their main advantage compared to software monitors is that they do not interfere with the system being measured. Hardware monitors are inherently limited to measuring information that can be interpreted at the hardware level without knowledge of operating system activities. Consequently, information like cpu service time, and file activity by process, cannot be obtained easily. Distribution estimates, e.g. mean channel service time, can be obtained for the system as a whole but not for specific work load classes. For these reasons hardware monitors are often

supplemented by software monitors or accounting data.

Logic state analysers used as hardware monitors (McKerrow 1983) provide very accurate microscopic measurement, e.g. instruction execution-path and execution-time. They can also be used for certain types of macroscopic measurement, e.g. process execution time or path, but again they lack operating system specific information. It is difficult to analyse some processes (e.g. the scheduler) without intimate knowledge of the current system load.

1.3.2. Software Monitors

In event-driven software monitors, significant events are defined (e.g. process switch), and the operating system is modified to record information about these events. Monitoring detail is limited by the number of events recorded. Consequently, if, after data analysis, it is found that a significant event has not been recorded then at best the session has to be rerun, and at worst the operating system has to be modified to record the event.

Sample-driven software monitors record data at the end of predetermined sample periods. The sample period is normally controlled by a timer. Thus, a time related sequence of states is recorded by a sample-driven monitor. Data collected during short measurement sessions can be recorded in memory, but normally disc file or tape storage is required.

An event-driven monitor has greater flexibility than a sample driven monitor but is likely to interfere with the system more. Depending upon the particular system and monitor, a software monitor may consume 20% or more of system resources (particularly cpu and channel time) and thus produce very questionable results. If this overhead can be kept to 5%, by appropriate event definition and probe implementation, software monitor results are reasonably accurate (Sauer and Chandy 1981).

In addition to interfering with system performance, software monitors have two other significant problems. First, the amount of data produced by an event trace may require excessive post-data reduction before meaningful results can be obtained. Secondly, software monitors must be specifically designed for the operating system and machine architecture. If not

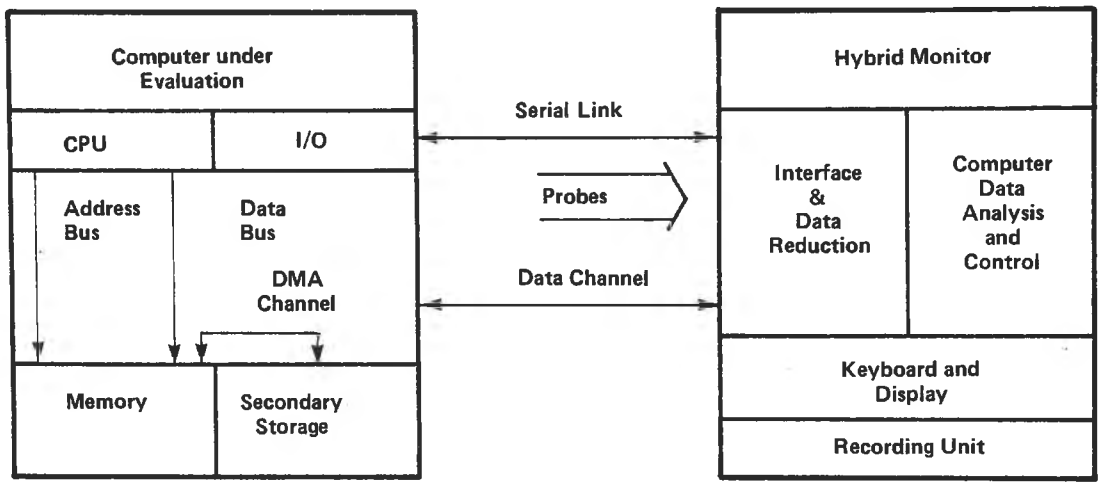


Figure 1.5 Basic Components of a Hybrid Monitor

designed into the system they can be difficult to add.

However, software monitors can get at operating-system-specific information (e.g. system queues) which hardware monitors cannot. Also, they can readily associate physical activity with logical entities (e.g. disc access with file name). In many ways, software and hardware tools complement one another. Hybrid tools have been developed to try to utilize the advantages of both.

1.3.3. Hybrid Monitors

Hybrid tools require the addition of both extra hardware and software to the system to be measured. They consist of external hardware tools which receive data collected by a software or firmware tool running in the system being measured (Svobodova 1973, Sebastian 1974). The hardware monitoring device is no longer invisible to the operating system, but it is treated as an "intelligent peripheral device" which can be used by a software monitor (Nutt 1975, Deutch & Grant 1971). Some even allow sections of the hardware tool, e.g. event counters, to be allocated to users under the control of software. A device of this type has been used to derive an on-line histogram of subroutine utilization and other similar tasks (Nemeth & Rovner 1971). Another type of hybrid monitor uses part of the existing hardware (a channel processor) as the monitor (Stevens 1968).

Burroughs have included a monitor micro instruction in the firmware of some of their computers (Wilner 1972, Denny 1977). This instruction writes a bit pattern, specified by the programmer, to pins accessible to the probes of a hardware tool. Nutt (1975) indicates that the system/monitor interface technique used by Hughes and Cronshaw (1973) and by Ruud (1972) best illustrated the trend, at that time, toward hybrid monitoring.

A hybrid monitor (figure 1.5) consists of a hardware monitor plus a data channel, between the monitor and the computer being evaluated, which can be used for transferring information between the two. While the hardware probes are used to monitor an event, the data channel can be used to obtain information about the stimulus of the event, thus, overcoming the

Figure 1.6 Common Measurement Methodology (Bell et al 1972)

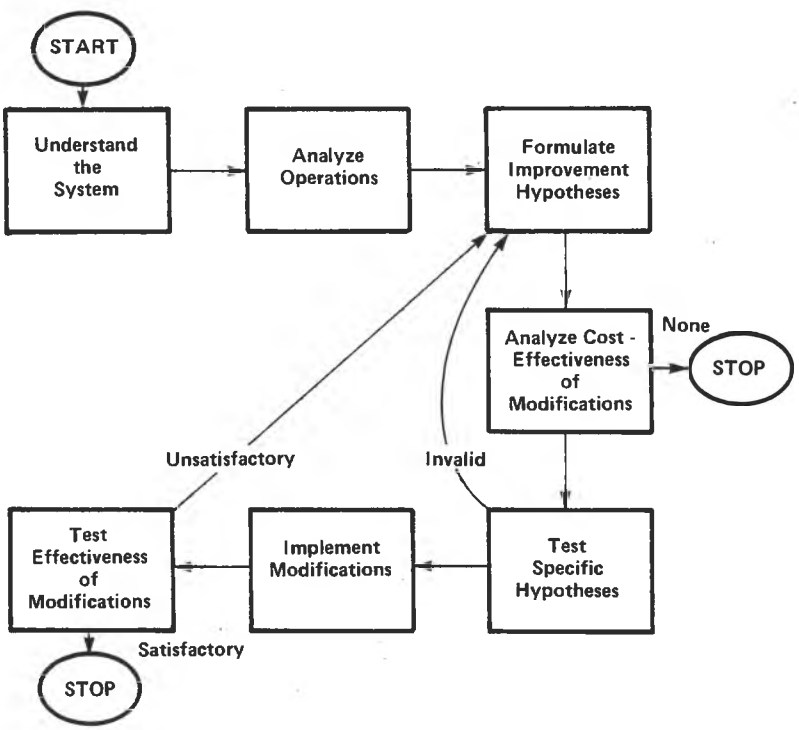


Figure 1.7 Proposed Performance Improvement Procedure (Bell et al 1972)

inherent limitation of hardware monitors. Alternatively, an event-driven software monitor can interrupt the external hardware-monitor and instruct it to record the required event-data, thus, reducing the overhead of the software monitor. The inclusion of computing power in the hardware monitor allows considerable real-time data reduction and analysis to be carried out, increasing the flexibility of the monitor.

Thus, hybrid monitors attempt to take advantage of the complementary nature of hardware and software tools. However, as Ferrari (1978a) comments: *the simultaneous usage of co-operating tools of the same or different nature, their coordination, and the partitioning of their functions and jurisdictions create problems which are still open to research.* One solution to these problems is discussed in chapter 6.

1.4. Measurement Methodology

Bell et al (1972) have suggested that many performance improvement studies produce very few results, for a high cost, because their measurement procedure resembles the flowchart of figure 1.6. Calingert (1967) identified the crux of the problem with his comment - *The key to performance evaluation as well as to systems design is to understand what systems are and how they work.* The purpose of measurement is to enable understanding. Measurement itself springs from an understanding of the problem at hand, and a knowledge of the basic principles of measurement. Nutt (1975) summarised it this way: *The most important questions to be asked before attempting to monitor a machine are what to measure and why the measurement should be made.*

This throws us right back to the scientific method. Modern science had its birth in the sixteenth and seventeenth centuries in the Christian civilization of Western Europe (Rhodes 1965). The world view of Aristotle - which included the ideas that nature was divine and in a sense self-explanatory, and that no phenomenon can be studied until its final cause has been established - was rejected. In its place, the sixteenth and seventeenth century scientists accepted the Christian world view, that because the universe was created by a God of order and purpose

then nature must also be ordered, and therefore able to be studied and described in an ordered manner. The basic presuppositions of science are:

- a belief in an orderly, regular, rational universe,
- a belief that this orderliness of the natural world is intelligible to the scientist,
- a belief in the reliability of human reason,
- a belief in a broad principle of causality, and
- a belief in the personal integrity of the scientist.

In these presuppositions the method of inductive and empirical enquiry, the process of observation, experiment and hypothesis, has its foundation. Hypothesis formulation involves the abstraction of certain elements from the total range of human experience. Thus, the scientific method is only one of a set of methods of describing human experience.

The popularizers of science have created a vast gulf in the minds of many between science and Christianity. One of the myths of scientific ideology is: since the empirical results of science have justified the validity of its basic assumptions no other foundation is needed. The early scientists made no such claim, rather they justified their assumptions on the basis of their belief in the biblical doctrine of creation by a personal, rational God.

A number of researchers have suggested a formalised framework for performance evaluation studies based upon the scientific method. The most comprehensive of these, proposed by Bell et al (1972) as a performance improvement procedure, is illustrated in figure 1.7. Measurement is done to provide data to test hypotheses about the system, program, etc. under study.

In concentrating on the measurement side of performance evaluation it is difficult, and unwise, to discuss measurement as an entity unrelated to other areas of performance evaluation. When measurement is constrained by other areas of performance evaluation these constraints are discussed. In the next chapter, a formulation of measurement is developed. The formulation defines the measures which can be made of a system, program, etc. during execution.

Which of these measures are to be made during a performance evaluation experiment depends upon the hypothesis to be tested. A methodology for making measurements to test hypotheses is described in Chapter 6.

2. A Formulation of Performance Measurement

The purpose of measurement
is insight not numbers.

Paraphrased by
Hamming (Cantrell & Ellison 1968)

The measurement of computer systems is almost as old as computer programming. Calingert (1967) wrote the first survey article, and Miller (1972) published the first bibliography of the field. Measurement techniques, and tools, are well documented in a number of books (Drummond 1973, Svobodova 1976a, Ferrari 1978a). Yet, the field remains a collection of ad hoc procedures with no unified body of knowledge. Dumont (1978) suggested: *the problem is that there is no general, overall context within which measurement and evaluation can take place.*

Browne and Shaw (1981) claim that the whole field of software engineering, performance measurement included, lacks a scientific foundation. They recommend that future experimental work be based on the traditional principles of science, where the first considerations are principles which are invariant across all software, and across hierarchies of abstract models. One goal of this chapter is to develop a set of measures-of-performance which apply at all levels in a computer system hierarchy.

Curtis (1980) suggests that progress in a scientific basis for software engineering depends upon improved measurement of fundamental constructs. In his opinion, a model of relationships among constructs becomes a theory when at least some constructs can be operationally defined in terms of observable data. Measurement does not define a construct, rather it quantifies a property of a construct. The major goal of this chapter is to develop a formulation of performance measurement from a model of a software object during execution, by defining the model constructs in terms of measurable data. A formulation helps us to concentrate on what we ought to measure, rather than on what is easy to measure.

In the last few years, the majority of theoretical work in the performance evaluation field

has been in the context of analytical queueing models (Spragins et al 1980). These models are mathematically tractable, have met with considerable success and thus, they are attractive to theorists and practitioners alike. Also, they can be used during the design and development stages of a system where measurement is not feasible (Sauer and Chandy 1981).

During the same period, other areas of Computer Science theory have made considerable progress, starting with McCarthy's (1962) pioneering work on a mathematical science of computation. Hoare (1969) has given us an axiomatic basis for programming; Manna (1974) has given us a mathematical theory of computation; Dijkstra (1976) produced a discipline of programming; de Bakker (1980) has developed a theory of program correctness; etc. Wand (1980), in the introduction to his text, comments: *If the past decade's work has one central thesis, it is that a program is a mathematical object... We therefore study the mathematical properties of programs. We mathematically prove the correctness of programs. We also use mathematical techniques to help us write programs.*

The thesis of this chapter is, because programs ~~are~~ can be modeled as mathematical objects we can not only measure their performance, but we can describe those measures mathematically; hence a formulation of performance measurement. On the basis of this formulation, performance measurement is codified into a unified body of knowledge. There has been considerable discussion, within the Faculty of Mathematical Sciences at the University of Wollongong, as to whether the material in this chapter is a theory of performance measurement ~~of~~ or a mathematical formalization of the measurements which can be made on an executing program. The question: What constitutes a theory in Computer Science? has been central to the discussion. In this dissertation, I have chosen to describe the content of this chapter as a formulation of performance measurement, and let the reader decide whether it is possible to have a theory of performance measurement or not.

2.1. Other Formulations of Measurement

By standing upon the shoulders of our forebears we can often see a little further than they, and sometimes a great deal further. The conceptual model of an executing program developed by Svobodova (1976a) and Ferrari (1978a) is expanded, and then generalised using Kolence's (1972) idea of a software unit.

Kolence's (1972) Software Physics is relevant to developing a formulation of performance measurement because it is the world-view from which he developed software performance-monitors. His concept is of a software physics which corresponds to natural physics at the basis conceptual level. He defined a basic constitutive definition for the nature of software: that of a software unit. From which he developed conservation laws and definitions for software energy, work, force and power. Kolence considered the computer performance measurement field to be best characterised as a set of numbers in a desperate search of unifying principles. He was concerned with the question: *How is meaning derived from a set of observations?* Software Physics was proposed as a way of finding that meaning.

Kolence gives some experimental data to support his formulation, but others seem not to have been convinced, as they have not taken up this methodology. The idea of a software physics deeply related to natural physics has not been proven. Software physics deals only with capacity management, and thus, is not a generalised formulation.

Ferrari (1978a) and Svobodova (1976a) have both developed formulations of program performance based upon similar models of program execution. The performance measurement formulation developed in this chapter is an expansion of both these formulations. Kolence's idea of a software unit (called an object in this thesis) has been used to generalise the formulation. Ferrari, Svobodova, and this formulation use the same model of program execution, but look at it from different perspectives. In each case, the concepts of: time sequence of events, state changes, and activity between events (modules) are present. Ferrari emphasises the time sequence of states and pays little attention to the activity between states, on the assumption that

significant activity will be reflected by the occurrence of a new state. Svobodova emphasises the occurrence of events, as these indicate the start and end of activities, on the assumption that we are interested in how often each activity occurs and for how long. This formulation emphasises the modules (activity between events); using the events to detect initiation, pausing, resumption, and termination of modules; and using the state (stimulus information) to give meaning to the actions of the modules. The perspective is, we want to know why execution has occurred the way it has, in addition to measuring the various indices.

2.2. World View

Although my definition of an object is similar to Kolence's software unit, the formulation developed in this chapter is based upon a different perspective. Trying to express one branch of science with the terminology and principles of another branch of science can place a straight jacket upon one's thinking process. While the natural creation exhibits considerable order, the creator has shown great variety in the principles underlying his designs. The Natural Scientist studies the creation and seeks to develop mathematical descriptions (abstractions) of that reality. The Computer Scientist studies the information handling processes designed into creation, seeks to develop algorithms to describe that reality, and then to code the algorithms into computer understandable form. Thus, algorithms are an abstract description of reality and programs are an implementation of that abstraction.

When we measure the performance of a program, we are measuring the performance of an implementation of an abstract description of a natural process. Ideally, we should be able to evaluate the performance of our implementation by comparing it to the natural reality. Unfortunately, this is not easy because often we cannot identify, and less often measure, a natural realisation of the process in creation. This may be possible when we understand the human brain more fully. Hence the difficulty of selecting and quantifying performance indices for any measurement study. Performance indices tend either to be based upon the performance of existing programs, or upon a theoretical expectation of performance. For example, algorithms

of order n are considered to be efficient and those of order n^2 are not, where n is the number of executions of a significant operation (presumably the operation which takes the most time).

The formulation developed in the next sections arose from thinking and praying about the practical problems involved in measurement with a hardware monitor (McKerrow 1983) and the subsequent design of a hybrid monitor (McKerrow 1984). In later chapters, the formulation is applied to a number of performance measurement issues, demonstrating its ability to codify the field into a unified body of knowledge.

2.3. Performance

Performance is what makes an object valuable to its user (Ferrari 1978). An object (O) is of value ($\mathcal{V}$) to its user if it performs its functions correctly ($\mathcal{C}$) and it performs them well ($\mathcal{P}$).

$$\mathcal{V} = f_1(\mathcal{C}, \mathcal{P}) \qquad 2.1$$
$$= value$$

How well an object performs is determined by evaluating its measured performance ($\mathcal{P}_m$) relative to predetermined performance indices ($\mathcal{P}_i$).

$$\mathcal{P} = \frac{\mathcal{P}_m}{\mathcal{P}_i} \qquad 2.2$$
$$= performance$$ - See Errata point 3.

$$\mathcal{C} = \frac{\mathcal{C}_m}{\mathcal{C}_i} \qquad 2.3$$
$$= correctness$$

Thus, performance evaluation seeks to ascertain the value of an object. Performance evaluation involves performance measurement which seeks to ascertain the extent ($\mathcal{E}$) of the object. This is a dynamic measurement of the object in execution not a static measurement of the text.

2.4. Object Definition

An **object** (O) is the particular entity whose performance we wish to measure. Each object has a function and a context. The **function** ($\mathcal{F}$) of an object is the set of transformations ($\mathcal{T}_d$) the

(a) Lower case

f	mathematical function
r_{zk}	module-to-path execution time ratio
t	time

(b) Subscripts

i	performance index
m	measured value
t	time

(c) Upper Case

$\mathscr{C}$	correctness of object
$\mathscr{D}$	values assigned to variables
$\mathscr{E}$	extent of object
$\mathscr{F}$	function of object
$\mathscr{H}$	execution history
$\mathscr{K}$	context of object
$\mathscr{M}$	meaning of object
$\mathscr{P}$	performance of object
$\mathscr{S}$	state of object
$\mathscr{T}$	transformation
$\mathscr{V}$	value of object

Table 2.1 Symbols Used and their Meanings - Script

object performs on the set of input data (I_d) to produce the set of output data (R_d, results), i.e. the purpose of the object.

$$\mathcal{F} = f_2(O) = \mathcal{T}_d \qquad 2.4$$
$$= \textit{set of transformations}$$
$$\mathcal{T}_d : I_d \rightarrow R_d \qquad 2.5$$
$$= \textit{set of output data}$$

Context ($\mathcal{K}$) is defined as the parts directly before and during an object that influence its meaning, i.e. the environment in which the object exists. The parts directly after an object can help us determine the meaning of the object, but don't, with the possible exception of backtracking, influence the meaning of the object. Thus, context includes the input data (which are the results of the parts before) and external events (E_e) which occur during the execution of the object and have some influence on it.

$$E = E_e + E_i \qquad 2.6$$
$$= \textit{set of events}$$
$$E_i = f_3(I_d, \mathcal{T}_d) \qquad 2.7$$
$$= \textit{set of internal events}$$
$$\mathcal{K} = E_e \cup I_d \qquad 2.8$$
$$= \textit{context}$$

Meaning ($\mathcal{M}$) is the reason for the actions of the object, and is related to the data the object is transforming.

$$\mathcal{M} = f_4(\mathcal{K}, \mathcal{T}_d) \qquad 2.9$$

2.5. Object Hierarchy

An object consists of a piece of executable code and its interactions with other pieces of code, with the computer, with peripheral devices, and with the user. An object can be as large as a complete system or as small as a single instruction. Normally, in the tradition of good programming practice, an object will have one entry point and one exit point. However, some systems are so unstructured that this may not be achievable.

An object at one level is defined as a set of objects at a lower level. For the sake of

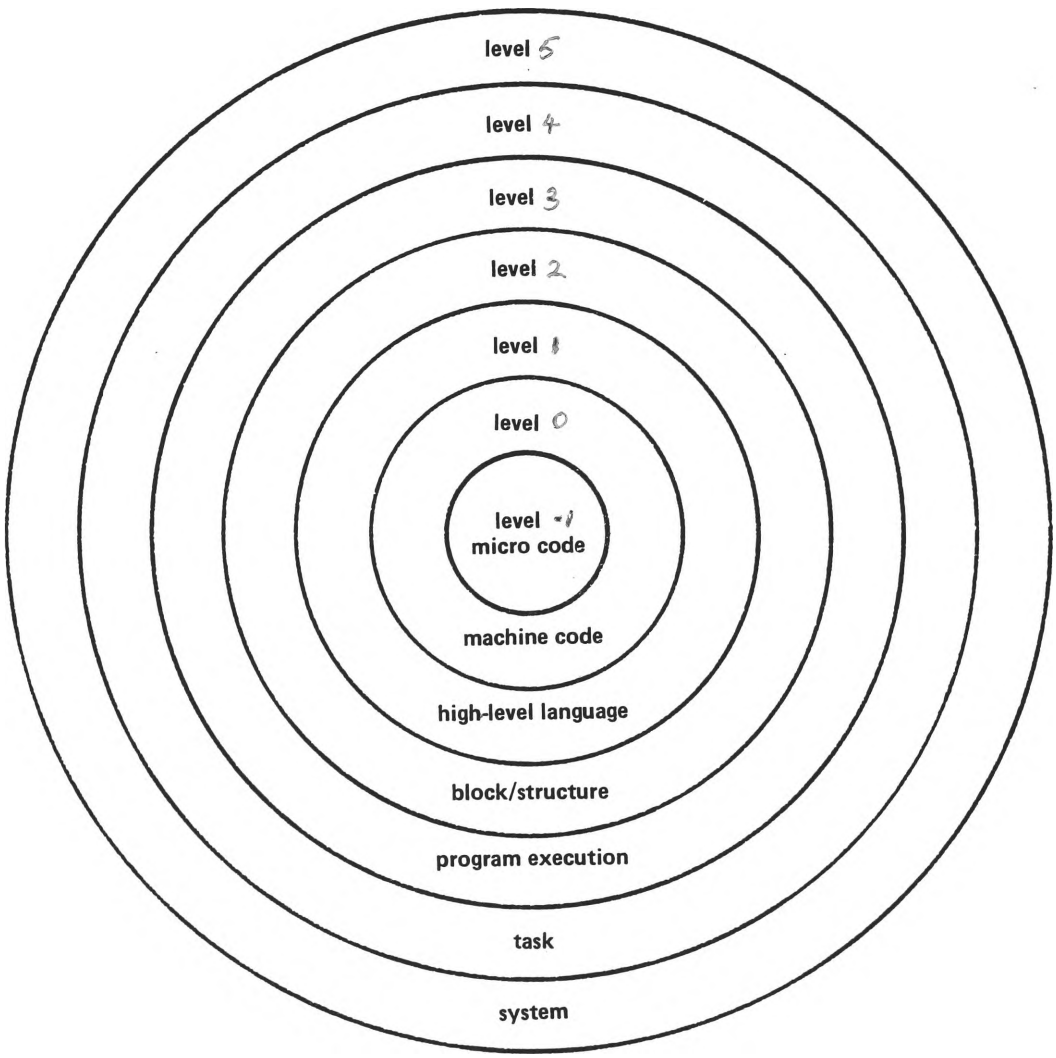


Figure 2.1 Object Hierarchy

clarity objects one-level-lower than the object-being-measured are called modules. The ordering of the modules in the object is not important, but is normally the order in which the modules occur in the source listing. A **module** (m) is a contiguous sequence of operations that occur in response to an event. An **event** is any action that initiates a significant change in the state of the object.

$$O_n = \left\{ O_{n-1,1} \,..\, O_{n-1,m} \right\} \qquad 2.10$$

An object at level n = set of objects at level n-1. See Errata point 1.

$$O_n = \left\{ m_1 \,..\, m_m \right\} \qquad 2.11$$

$$where\ m_1 = O_{n-1,1}\ etc. \qquad 2.12$$

Thus, a computer system can be viewed as a hierarchy of objects where a higher-level object consists of a set of lower-level objects (figure 2.1). For a detailed theoretical treatment of the fundamental proposition that complex systems are often hierarchic, and that hierarchical systems are often nearly-completely decomposable, see the monograph by Courtois (1977). The complete computer system is considered to be the super set. Objects of level zero (the lowest level) will normally be considered to be the set of individual machine-code-instructions, or subsets of machine instructions, although for some measurements it may be desirable to go one step lower to the microcode. On most modern microprocessors, and minicomputers, machine-code instructions are indivisible, but on some machines machine-code instructions can be interrupted. In the latter case it may be necessary to consider the microcode as the lowest level. The set of machine-code-instructions has been chosen as the lowest level because:

- the machine-code defines the computer architecture exactly (from the programmer's point of view),
- at this level everything can be measured, and
- in most machines, particularly microcomputers, it is not possible to go any lower.

Thus an object at level zero is a set of machine-code-instructions. The set can range from a single instruction to the complete set of machine-code-instructions. A module of an object at level zero is an individual instruction.

$$M(O_0) = True \qquad 2.13$$

where M is the predicate on the set of objects which is true if and only if that set is measurable.

We can now hypothesise, that given a set of functions which map from one level of the hierarchy to the next higher level, then an object at any level is measurable.

Principle
~~**Theorem**~~ **2.1**

An object that can be obtained recursively from measurable objects is itself measurable.

Note the similarity to the computability theorem (Cutland 1980, theorem 4.4): *a function obtained recursively from computable functions is itself computable.*

We have already stated that the objects at level zero are measurable (equation 2.13). If we assume that the objects at level n are measurable then can we prove by induction that the objects at level n + 1 are also measurable. When we look at the set of measurements that can be made on an object we will see that they are:

(a) defined in terms of measurements made at a lower level, and

(b) composed of lower-level measurements in such a way that data reduction occurs.

Thus, a measurement at level n + 1 is composed of measurements made at level n.

$$M(O_n) = True\ for\ all\ n \qquad 2.14$$

Kolence expressed it this way: *any truth we can say about software units [objects] must be universally true from the level of the single instruction up to the set of all software ever produced.* The claim is: the recursive nature of an object leads to a generalised formulation of performance measurement which applies at all levels of the object hierarchy.

A significant problem in performance evaluation has always been the reduction of the

enormous amounts of data available at level zero into a tractable set of meaningful data as we move up to higher levels. A second claim is: that the recursive nature of the object definition leads to natural methods of deducing high level quantities from low level data, and natural methods of data reduction.

Principle
~~Theorem~~ **2.2**

Measurements of an object, that has been obtained recursively from other objects, can also be obtained recursively from the measurements of those objects.

Principle
~~Theorem~~ **2.3**

Measurements made at one level in an object filter out lower-level values, and hence lower-level measurements cannot be obtained from higher level measurements.

Thus measurement tools selected for a particular level will automatically filter out lower-level data.

Principle
~~Theorem~~ **2.4**

Objects of interest in performance evaluation fit into the following hierarchy (figure 2.1).

- System level - computer system and users
- Task level - a group of processes which execute to perform a request
- Program-execution level - individual programs, processes, sub-programs, procedures, functions
- Block/Structure level - sequential blocks of code, compound statements, small procedures, functions.
- High-level-language level - individual instructions, interpreter calls
- Machine-Code level - assembler instructions, memory bus cycles

- Microcode level - microcode instructions and ALU cycles.

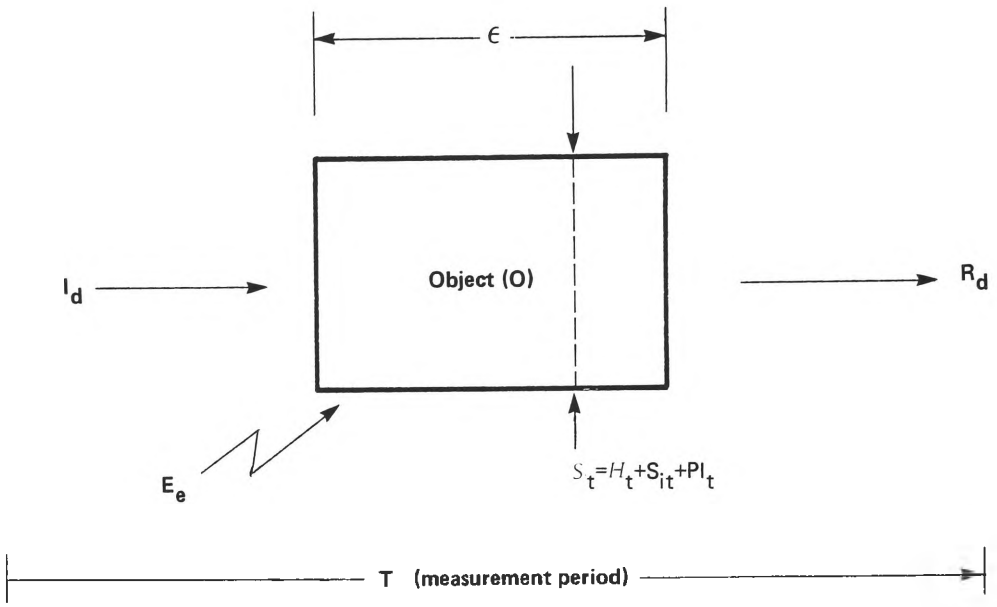
In some situations not all levels of the hierarchy may exist. For example, a system programmed in assembler will not include the high-level language level but will go from machine-code level to block-structure level.

2.6. Performance Measurement

The process of program design is best tackled by the process of stepwise refinement (Wirth 1971) where a problem is simplified by decomposing it into smaller problems. This process of decomposition is repeated until the problems are intellectually manageable, and detailed solutions can be studied. In a similar way, the hierarchical definition of an object is a means of decomposing one large measurement problem into a set of smaller measurement problems. This process of decomposition is repeated until the level in the hierarchy is reached where measurements can be made with the available tools. In the discussion that follows, the level at which measurements are made is considered to be the module level.

At all levels of the performance-evaluation hierarchy, performance measurement consists of ascertaining the extent of an object and the extent of the set of modules contained in the object. Before the extent of an object during execution can be defined, **a conceptual model of an executing object** is required. An object during execution is an ordered sequence of modules, where the order is the sequence in which execution occurs. This sequence is called an **execution path**. An object has one or more execution paths. Associated with each execution path is an **execution time**: the time taken to execute the object, which is the sum of the times taken to execute the modules in the sequence.

The termination of one module, in the sequence, and the start of the next is caused by an event. Detecting events, with a measurement tool, enables the decomposition of an object into modules. Associated with each event is a set of **stimulus information** (state, etc.). The stimulus information gives meaning to the actions of the object i.e. it enables the analyst to determine why the object behaved the way it did.



Object Bounds

start of execution
pause in execution
resumption of execution
termination of execution

Calculated Values

set of object execution frequencies
set of module execution frequencies
set of relative path execution times
set of relative module execution times
set of path utilizations
set of module utilizations
object state at time t
object throughput
module throughput
module-to-path execution time ratio

Measured Values

set of execution paths
measurement period
set of path execution times
set of execution path counts
total number of object executions
set of module execution counts
set of module execution times
set of module execution paths
total number of module executions
stimulus information
program counter at time t
execution history at time t
memory usage
quantity of data
job class

Figure 2.2 Graphical Representation of an Object (Symbols and their meanings are given in tables 2.1 to 2.3)

A monitoring tool should record an **event trace** of modules, and associated stimulus information. The event trace is a sequence of tuples containing a module identifier and the time, relative to the start of the measurement period, at which the module commenced execution. When stimulus information is recorded, a copy of the current-event-trace tuple (current module identifier, time of recording of stimulus information relative to the start of the measurement period) is included in the stimulus record. From these records, the extent of the object is obtained.

To record an event trace, a monitoring tool must be able to detect the bounds of the modules in the object under study. These bounds include the start of execution of a module and the termination of execution of the same module. Termination of a module is often, but not always, indicated by the start of the next module in the sequence. In addition, external events can cause a module to pause, while another module is executed, and then resume. The tool must also be able to detect the pausing and resumption of a module.

From the recorded information, the following parameters, measured values and calculated values (figure 2.2), are determined.

a. **Execution Path**

What path was taken through the object during its execution? At the machine-code level, an execution path is defined as one of the set of machine-code instructions. An object at any level may have several execution-paths, where each individual execution-path (p_k) is an ordered set of modules.

$$p_k = \left\{ m_1 \,..\, m_2 \right\} \tag{2.15}$$

and

$$p_k \subseteq O_n \tag{2.16}$$

The set of modules that make up an object is partially ordered, in that the order represents only one of the possible execution paths. The set of execution paths (P_{en}) for an object at level n

includes all the possible execution paths through the object.

$$P_{en} = \left\{ p_1 .. p_k \right\} \qquad 2.17$$

During a measurement period (T), each path is executed a number of times (c_k) giving a set of path execution counts (N_{cn}).

$$N_{cn} = \left\{ c_1 .. c_k \right\} \qquad 2.18$$

We need to remember that each module in an execution path is an object at the next-lower-level, with its own set of execution paths (P_m) (at the machine-code level there is only one member in the set). This complexity is a consequence of the recursive definition of an object, and methods of handling it will be discussed later. However, a significant reduction in the volume of data from that available at the machine-code level has already been achieved.

b. **Execution Time**

How long does an object take to execute? Only in the cases where there is a single execution-path will there be a single execution-time. There are three types of execution path where this will occur: an individual, non-branching instruction, a sequential block of such instructions, and a loop with a fixed loop-count. In all other cases there will be several execution times for an object. Situations where one path can have a range of execution times are discussed in section 6.1.2.

An object at level n has a set of execution times (T_{en}) which consist of an execution time (t_k) for each execution path (p_k). The execution time of a path (t_k) is the sum of the execution times (b_{mq}) of the modules in the path.

$$T_{en} = \left\{ t_1 .. t_k \right\} \qquad 2.19$$

for the set of execution paths P_{en}

The execution time of the module (b_{mq}) is one of the set of possible execution-times for the

module (T_m). As only one path is executed each time the module is executed, only one module execution-time contributes to the path execution-time for that module invocation.

$$T_m = \left\{ b_{m1} .. b_{mq} \right\} \qquad 2.20$$

where there are q paths through module m

If a module has only one execution path then the set T_m normally reduces to a single member. Exceptions to this are discussed in section 6.1.2. The set of individual-module execution-times is a member of the super set of module execution-times (T_{bn}).

$$T_{bn} = \left\{ T_1 .. T_m \right\} \qquad 2.21$$

for the set of modules in O

c. ~~Frequence~~ Frequency **of Execution**

How often is each path through an object executed during the measurement period? How often is each module in the object executed during the measurement period? The measurement period (T) is specified in units of time, and normally corresponds to the time taken to complete a certain experiment rather than a fixed time-interval. An object at level n has a set of execution frequencies (F_{en}) which consist of the number of executions (c_k) for each path (p_k) executed during the measurement period divided by the total number of executions (N_e) of the object.

$$f_k = \frac{c_k}{N_e} \qquad 2.22$$

$$= \text{~~frequence~~ frequency of execution of } p_k$$

$$F_{en} = \left\{ f_1 .. f_k \right\} \qquad 2.23$$

for the set of execution paths P_{en}

$$N_e = \sum_{j=1}^{k} c_j \qquad 2.24$$

An object at level n is a set of m modules at level n-1 (equations 2.11, 2.12). This object has a

(a) Lower case

a_m	number of times module m was executed
a_{mq}	number of times path q of module m was executed
b_{mq}	individual module execution time
c_k	path execution count for path k
d_{mq}	execution time of path q in module m
e_m	execution frequency of module m
f_k	execution frequency of path k
m_m	module m
p_k	object execution path k
r_k	relative path-execution-time
s_k	set of stimulus information for path k
s_m	set of stimulus information for module m
t_b	maximum module execution time
t_k	execution time of path k
t_p	maximum object execution time
u_k	path utilization
v_m	module utilization

(b) Subscripts

d	data
e	external
i	internal
k	object path number
m	module number
n	object level
q	module path number
z	position of module in object path

Table 2.2 Symbols Used and their meaning - lower case printed

set of module execution-frequencies (F_{mn}). The module-execution-frequency (e_m) for module m is the number of times module m was executed (a_m) during the measurement period divided by the total number of module executions (N_m).

$$N_m = \sum_{j=1}^{m} a_j \qquad 2.25$$

$$e_m = \frac{a_m}{N_m} \qquad 2.26$$

$$\text{and } N_a = \left\{ a_1 .. a_m \right\} \qquad 2.27$$

$$F_{mn} = \left\{ e_1 .. e_m \right\} \qquad 2.28$$

for the set of modules in O

d. **Throughput**

Is the object executed frequently? Object throughput (N_o), the number of object executions per unit time, is calculated by dividing the number of executions of the object (N_e) by the measurement period (T).

$$N_o = \frac{N_e}{T} \qquad 2.29$$

Module throughput (N_t) is the number of module executions per unit time.

$$N_t = \frac{N_m}{T} \qquad 2.30$$

c. **Relative Execution-Times**

Does one path take considerably longer to execute than other paths do? Does one module take considerably longer to execute than other modules do? An object at level n has a set of normalised relative-path-execution times (T_{rn}) which are derived by dividing the execution time for each path (t_k) by the maximum execution time for any path (t_p).

$$t_p = \max(T_{en}) \qquad 2.31$$

$$r_k = \frac{t_k}{t_p} \qquad 2.32$$

$$T_{rn} = \left\{ r_1 .. r_k \right\} \qquad 2.33$$

for the set of execution paths P_{en}

An object at level n has a set of normalised relative-module-execution-times (T_{sn}) which are derived by dividing each member of the set of execution times for each module (b_{mq}) by the maximum execution time for any module (t_b).

$$t_b = \max(T_{bn}) \tag{2.34}$$

$$d_{mq} = \frac{b_{mq}}{t_b} \tag{2.35}$$

$$T_{sn} = \left\{ d_{11} \,..\, d_{mq} \right\} \tag{2.36}$$

for the set of modules in O

Each path through an object has a set of module-to-path-execution-time ratios ($E_{\wedge}$) which are calculated by dividing the execution time of each module (b_{mq}) in the path (P_k) by the execution time for the path (t_k).

$$r_{zk} = \frac{b_{mq}}{t_k} \tag{2.37}$$

$$E_{\wedge} = \left\{ r_{1k} \,..\, r_{zk} \right\} \tag{2.38}$$

where there are z modules in the path

f. **Utilization**

Does the object spend most of its time executing one path? Does the object spend most of its time executing one module? An object at level n has a set of path utilizations (U_{pn}) which are the product of the path execution time (t_k) and the number of times each path is executed (c_k) divided by the measurement period (T).

$$u_k = t_k \times \frac{c_k}{T} \tag{2.39}$$

$$U_{pn} = \left\{ u_1 \,..\, u_k \right\} \tag{2.40}$$

for the set of paths P_{en}

The number of times a module was executed (a_m) is the sum of the number of times each path (a_{mq}) in the module was executed, during the measurement period. An object has a set of module utilizations (U_{mn}) which are the total execution time for each module divided by the measurement period.

$$v_m = \frac{\left(\sum_{i=1}^{m} a_{mq} \times b_{mq} \right)}{T} \qquad 2.41$$

In the case where there is only one path through a module, this equation reduces to

$$v_m = b_m \times \frac{a_m}{T} \qquad 2.42$$

$$U_{mn} = \left\{ v_1 \,..\, v_m \right\} \qquad 2.43$$

for the set of modules in the object O

g. **Stimulus Information**

Which information gives meaning to the actions of the object? Two overlapping sets of stimulus information exist in an object: the set of data used in decision making ($\mathcal{D}_d$), and the set of data being transformed by the object ($\mathcal{D}_t$).

$$S_{in} = \mathcal{D}_d \cup \mathcal{D}_t \qquad 2.44$$

The set of stimulus information (S_{in}) for an object at level n is the union of the sets of stimulus information (s_k) for each path through the object, and also the union of the sets of stimulus information for each module (s_m) in the object.

$$S_{in} = s_1 \cup s_2 \cup \,..\, s_k = s_1 \cup s_2 \cup \,..\, s_m \qquad 2.45$$

From equations 2.8 and 2.9 we see that stimulus information includes: input data (I_d), output data (R_d), partially transformed data, and external events. Internal events are not included as they are a function of the input data and the transformations (equation 2.7), and they cause a change in state which is indicated by the execution path. The data used by the object when deciding which execution path to take is already included in the stimulus information. The first step in determining which stimulus data to record during a measurement session is to define the

set of data used in decision making.

One important use of stimulus information is workload characterisation (Ferrari 1972). When evaluating a system it is important to know what type of work load, and how much, the system is handling during the measurement period. The object (job) class (J_c) of an object (for example batch, interactive editing, Pascal compiler, etc.) is a piece of stimulus information that both describes an object (e.g. Pascal program), and is used in decision making (e.g. execute a Pascal program).

h. **Memory Usage**

How much memory is used by the object during execution? Simplistically, the memory usage (M_u) of an object during execution is the range of instruction accesses, i.e. the difference between the addresses of the lowest and highest locations from which instructions are fetched, plus the range of data accesses. In many situations, the above measure is too ~~course~~ coarse. Instructions may be loaded into discontiguous segments, or data may be both local to an object and common to a set of objects, leaving large holes in the apparent memory-usage that are not used by the object.

The measure of memory usage (M_u) of an object is refined to be the sum of the set-of-instructions-segments, where an instruction-segment is defined by a contiguous range of instruction accesses, plus the sum of the set-of-data-segments, where a data-segment is a contiguous range of data accesses. This measure may not be the same as a static measure of memory usage (M_s) obtained from the object text, because parts of the object may not be executed, or memory usage by the object may be dynamic. In the latter case, an accurate measure cannot be obtained from the text without knowledge of the stimulus information. The former case clearly indicates the possibility of a bug in the object.

At any time during the execution of an object, the memory usage may only be a portion of the total. For example, in a segmented system only some of the object may be in memory. It is in this context that measurements of the objects working set are made. This information is

obtained as stimulus information from the memory-allocation process.

In the introduction to a special issue of IEEE Computer on program behavior, Domenico Ferrari (1976) comments: *Among the performance characteristics of programs, the patterns of memory references they generate have the unique property of being totally irrelevant in a non-virtual memory context, and perhaps the most important aspect of program behavior in a virtual memory system.* Measurement of program behavior usually produces a reference string (address trace - section 3.8). From this string, the memory-referencing behavior of the program can be determined. This concept can be extended to a page-reference string, which can be used to analyse a programs working-set, and to a data-base-reference string, for the analysis of references to a data base (Rodriguez-Rosell 1976).

These ideas fit closely with our conceptual model of an executing object: an ordered sequence of modules. At the machine-code level, the reference string is the sequence of program-instruction references interleaved with the sequence of variable-access references. At a higher level, the reference string is the sequence of memory segments used by the object. Measurement of these reference strings is discussed in section 6.3, where the event trace is extended to include address-bus information.

i. **Data Structure Access**

What variables are accessed, and how often, during the execution of an object? During execution, an object will manipulate data. Data is stored in data structures ranging from simple variables, to arrays, to records, to stacks, to queues, to trees. Some of these are static, and can be mapped directly onto object text; others are dynamic, and depend upon the quantity of data (Q_d) being processed by the object, which is the sum of the quantities of data for each data structure in the set of data structures.

While the location of static data-structures can be found from the object text, usually only the initial element of a dynamic structure can be found from the text. Thus, measuring dynamic data-structures involves measuring the size of the structure (i.e. the memory usage).

E	set of events
E_r	set of module-to-path execution-time-ratios
F_{en}	set of object execution frequencies
F_{mn}	set of module execution frequencies
I	input to object
J_c	job class
M	measurable predicate
M_s	static memory usage
M_u	dynamic memory usage
N_a	set of module execution counts
N_{cn}	set of path execution counts
N_e	total number of object executions
N_m	total number of module executions
N_o	object throughput
N_t	module throughput
O_n	object at level n
PC_t	program counter at time t
P_{en}	set of object execution paths
P_m	set of module execution paths
Q_d	quantity of data
R	results out of object
S_{in}	set of stimulus information
T	measurement period
T_{bn}	super set of module execution times
T_{en}	set of object execution times
T_m	set of module-execution times
T_{sn}	set of normalised relative-module execution times
T_{rn}	set of normalised relative-path-execution times
U_{mn}	set of module utilizations
U_{pn}	set of object path utilizations

Table 2.3 Symbols Used and their Meaning - Upper Case Printed

This can be done either by measuring memory usage or by counting, and summing, the data manipulations performed upon the structure (module-execution counts).

Monitoring object access to static variables is usually only required for debugging purposes. In many real-time systems, communication between independent objects occurs through common data-structures. Processes causing data corruption can be pinpointed by monitoring common-data-structure accesses. Also, compiler and linker faults can be identified by monitoring variable accesses.

2.7. Object Extent and Object State

The actual object, and stimulus variables, to be measured depends upon the purpose of the measurement study, the level of the object in the hierarchy, the context in which the object is placed, and the function it performs. However, measurement of the above parameters defines the extent of the object during the measurement period (figure 2.2).

$$\mathcal{E} = \left\{ P_e, T_e, T_m, N_c, N_e, N_a, T_b, N_m, S_i, P_m, M_u, Q_d, J_c \right\} \cup \left\{ F_e, F_m, T_r, T_s, U_p, U_m, N_o, N_t, E \right\}$$

$$= \textit{measured values} \cup \textit{calculated values} \qquad 2.46$$

(these symbols are defined in tables 2.1-3)

The set of possible calculated-values, involving all possible combinations and permutations of measured values, is very large. The subset included here appears to include all those in common use. Some extensions to this subset move out of the field of performance measurement into the fields of performance comparison, for example comparing the execution times of a program running on different processors, and performance evaluation, for example cost effectiveness - the ratio of throughput to system cost. System Price, in dollars, is a value that needs to be ascertained, but it is not included in the measures because it only indirectly effects the extent of an executing object.

The state ($\mathcal{S}_t$) of an object at level n is the value of these parameters (measured and calculated) at any point in time (t) during the life of the object plus the values currently assigned to all data variables ($\mathcal{D}_t$) and the program counter (PC_t). Thus, the state is the execution history ($\mathcal{H}_t$) up to this point in time, plus the current status of all variables. The values assigned to the variables reflect the execution history, and the partial completeness of the transformations. The current execution-path and the point at which execution is occurring at time t is defined precisely by the program counter.

$$\mathcal{S}_t = \mathcal{H}_t + S_{i\,t} + PC_t \qquad 2.47$$

Regularly, the execution of an object is interrupted by an external event which causes that object to pause while another object is executed. If the data and the program counter portions of the state are saved then the object can recommence simply by restoring those values. The history portion of the status is not explicitly carried along with the object and must be recorded, as the object executes, by the measurement tool. Also, the measurement tool must be able to detect a pause in the execution of an object, in addition to the start and finish of object execution.

For some objects, measuring pauses in the execution of the object is an important part of measuring the object, and other objects, which execute during pauses in the object, are irrelevant. For example, when measuring a disc driver the time between a request to seek and a seek interrupt is an important parameter: the seek execution-time. In this situation, the pause can be classified as a hardware object with the execution path being the operation performed by the hardware. In other situations, a pause can be classified as a wait module with its execution path being the empty set. All the previous discussion applies to these modules also, providing a way of measuring software/hardware interactions.

2.8. Data Reduction and Analysis

The set of measurements that define the extent of the object (assuming at this stage that such measurements can be made) reduce the enormous quantity of data available to ordered sets of

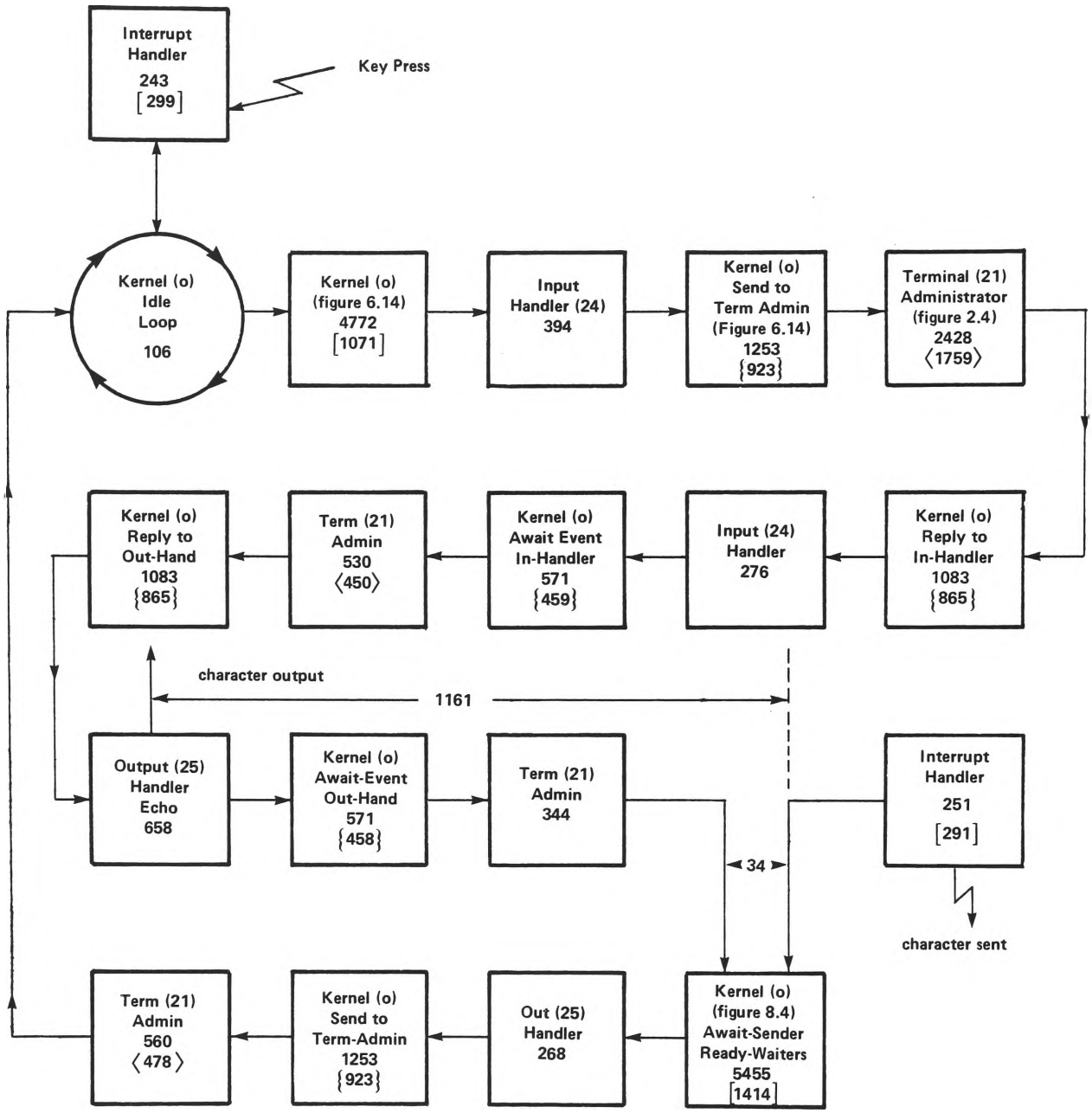


Figure 2.3 Event-Trace Graph showing the sequence of processes executed to handle the input of a simple character from a terminal keyboard (section 8.4) - all times in microseconds - numbers in brackets are process numbers.

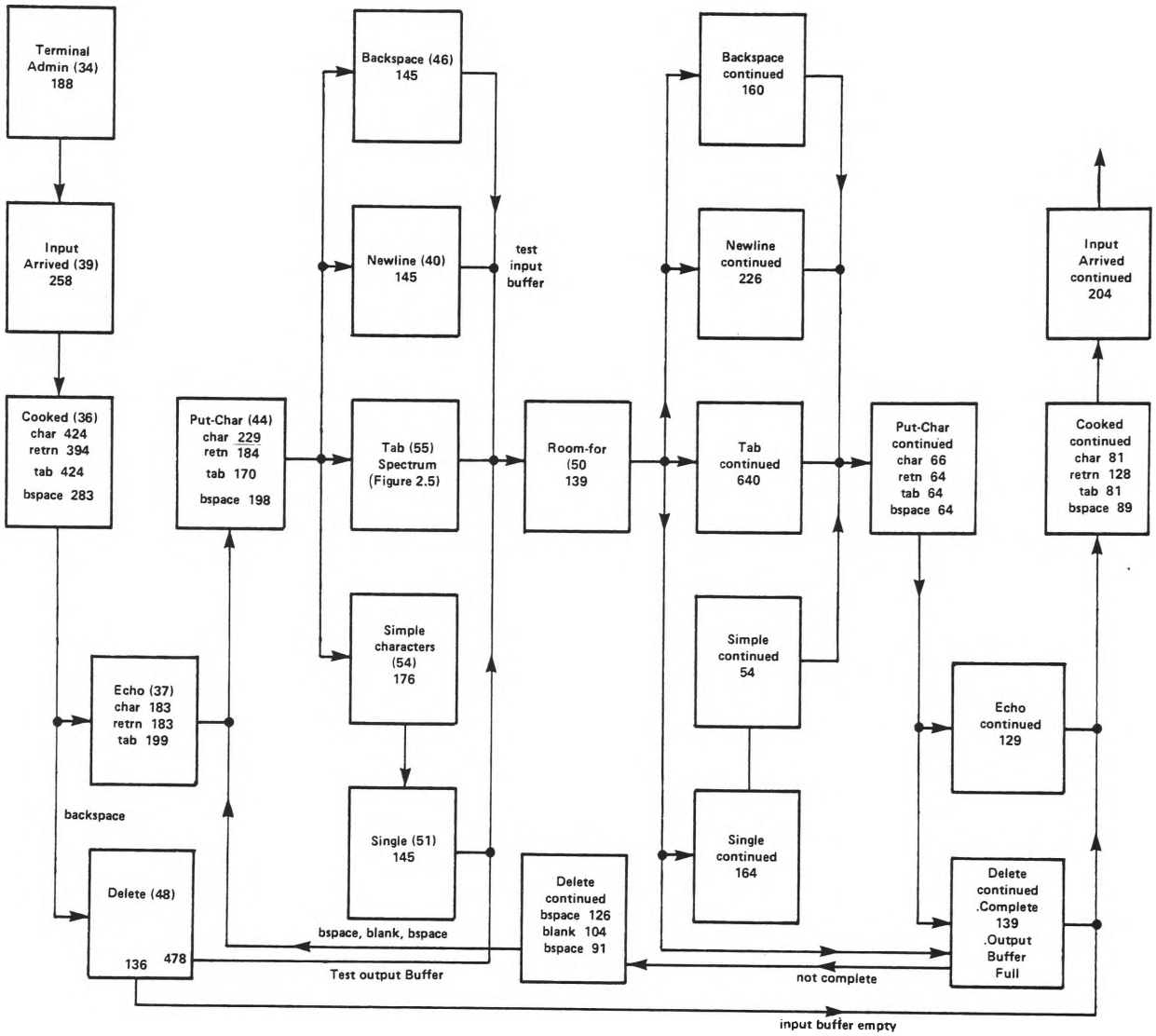


Figure 2.4 Execution-Path Flow-Graph showing the set of execution paths through the Terminal Administrator process (figure 2.3) - all times in microseconds - numbers in brackets are module numbers.

data for each measurement. For objects at or near the level the measuring tool is working at, these sets are small and tractable. As you move up the hierarchy, the volume of data grows rapidly, and consequently the difficulty of computing high-level data from low-level data increases rapidly. A number of data reduction, and analysis, methods are available to tackle this problem.

Traditional methods of data reduction involve selecting appropriate hardware, and software, tools for the level of interest. These provide data at that level, but lower-level data is lost. For example, a cpu-utilization hardware-tool measures the time the cpu is busy during the monitoring period, very accurately, and all other data is discarded. CPU utilization time is a measurement of the execution time of a system level object. Many software tools count the number-of-executions of particular objects, once again reducing the data to that required.

The measurements lend themselves to **graphical methods** of display and analysis (Ferrari et al (1983) illustrate a number of the commonly used data presentation methods). A number of these graphical methods are illustrated:

- An **event-trace graph** is a graphical display of the event trace, and stimulus information, with respect to a time axis (figure 2.3). This display allows the analyst to walk through the execution path of the object, and to study the conditions that caused that path to be taken.

- An **execution-path flow-graph** (figure 2.4) is a drawing showing one or more of the set of execution paths through an object. It is a directed graph of modules with the sets of execution times, or pointers to module spectrum, recorded in the nodes, and stimulus information recorded on the arrows. This graph gives a clear picture of the inter-relations between the modules in an object.

- An **object spectrum** (when referring to modules it is a module spectrum) is a plot of the number of executions of each path verses the execution time of the path, i.e. N_c v.s. T_e. A normalised object spectrum is the plot of the set of execution frequencies (F_e) versus

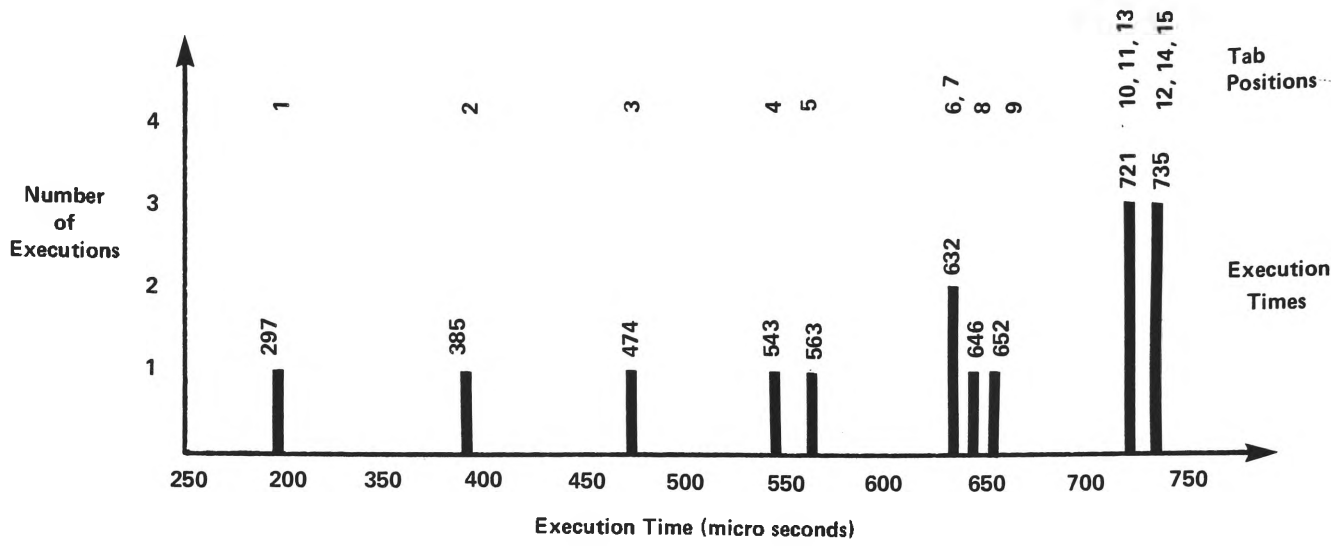


Figure 2.5 Object Spectrum for the Module Tab (figure 2.4) showing the execution time for the first 15 tab positions across the screen - all times in microseconds.

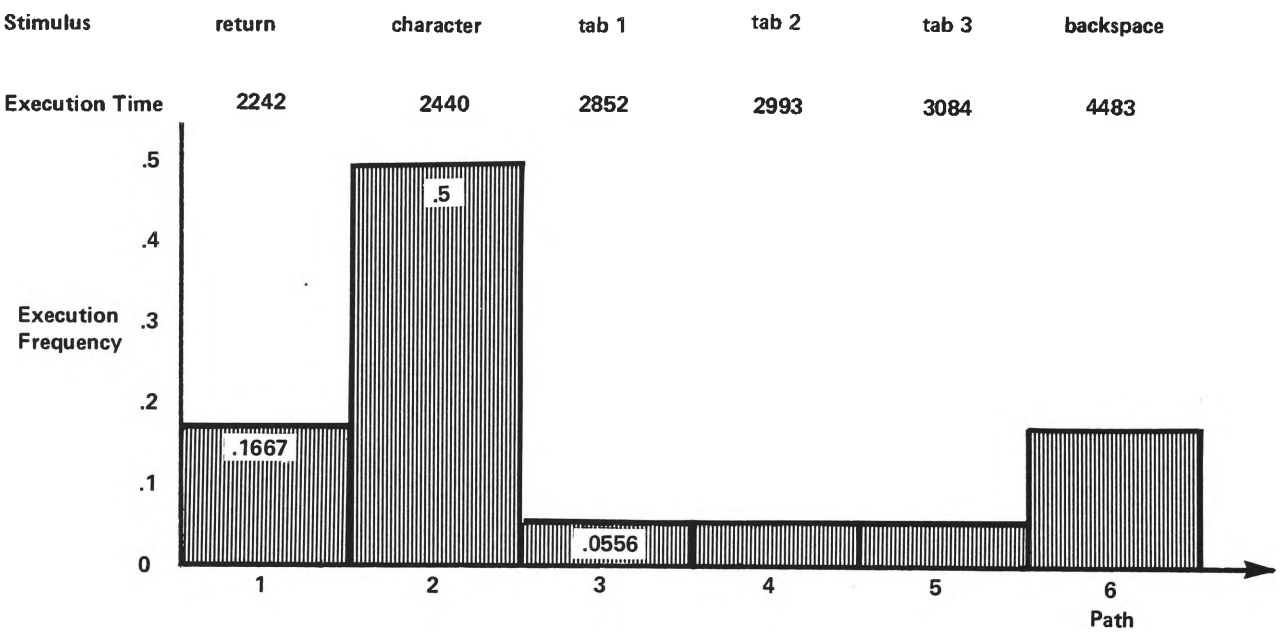


Figure 2.6 Execution-Path Profile for the Terminal Administration process (figure 2.4) for a variety of input characters (18 characters) - all times in microseconds.

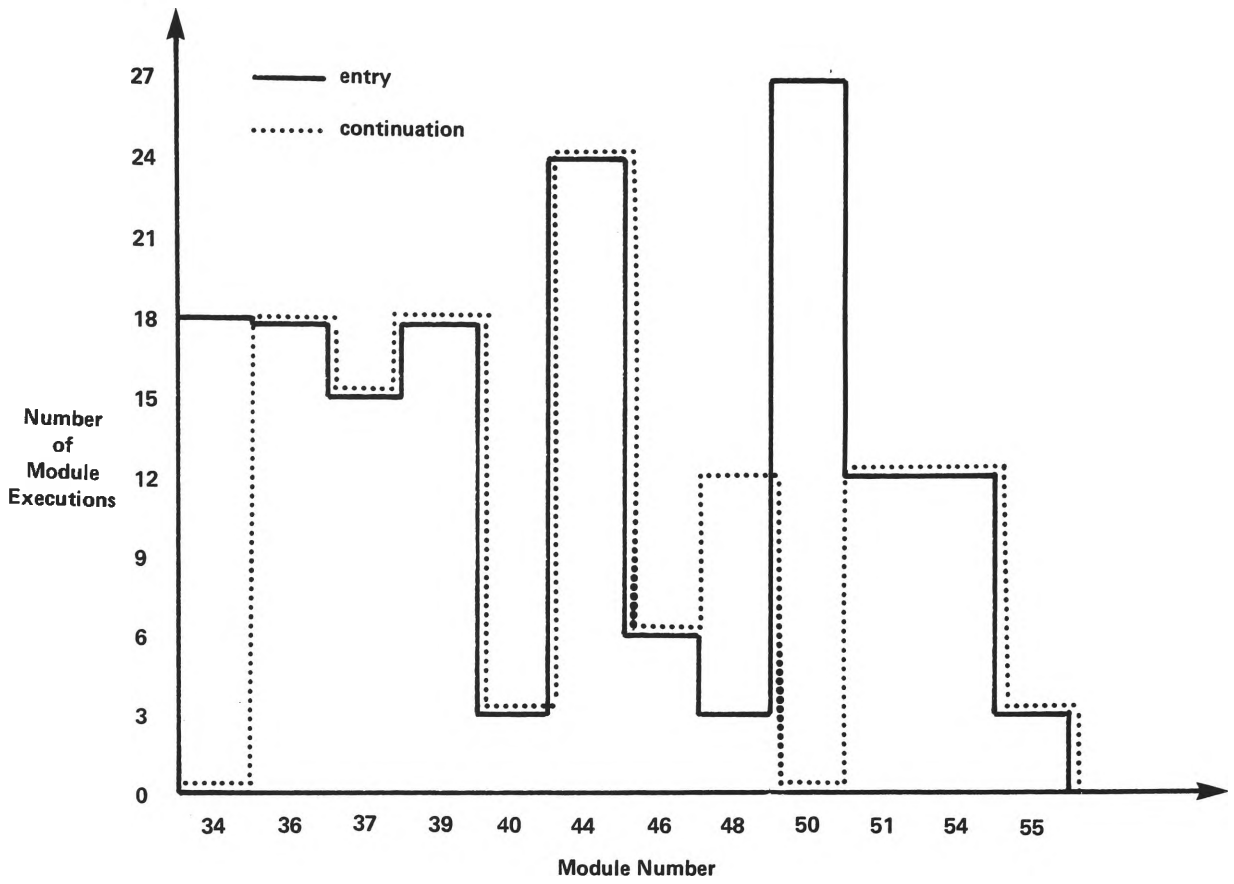


Figure 2.7 Module Profile for the set of 18 execution paths shown in figure 2.6 - module names are given in figure 2.4.

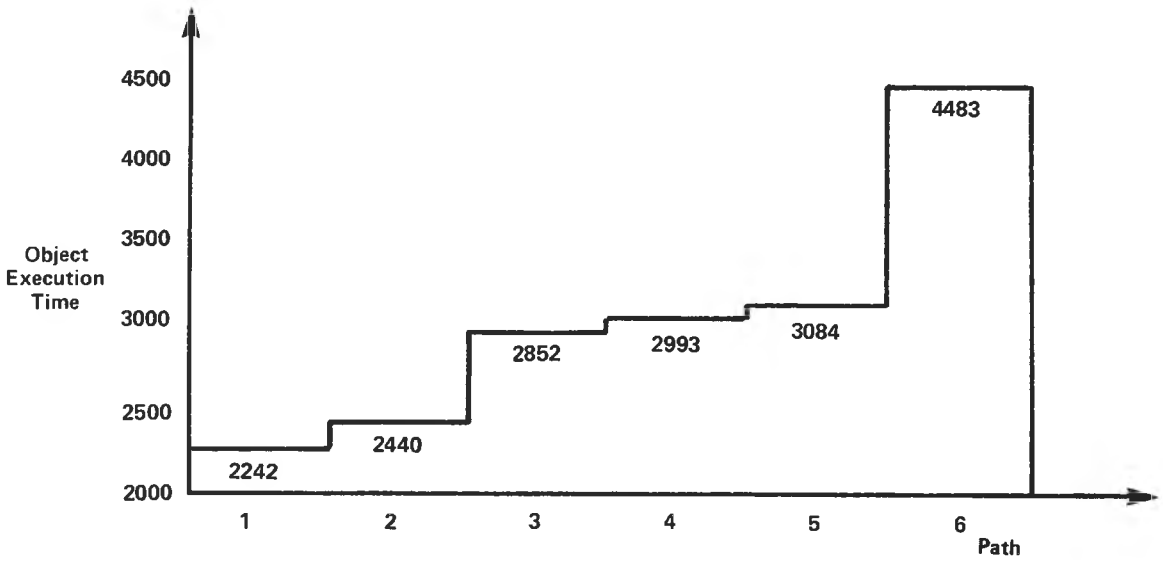


Figure 2.8 Path-Execution-Time Profile for the execution paths, shown in figure 2.6, of the Terminal Administrator process - all times in microseconds.

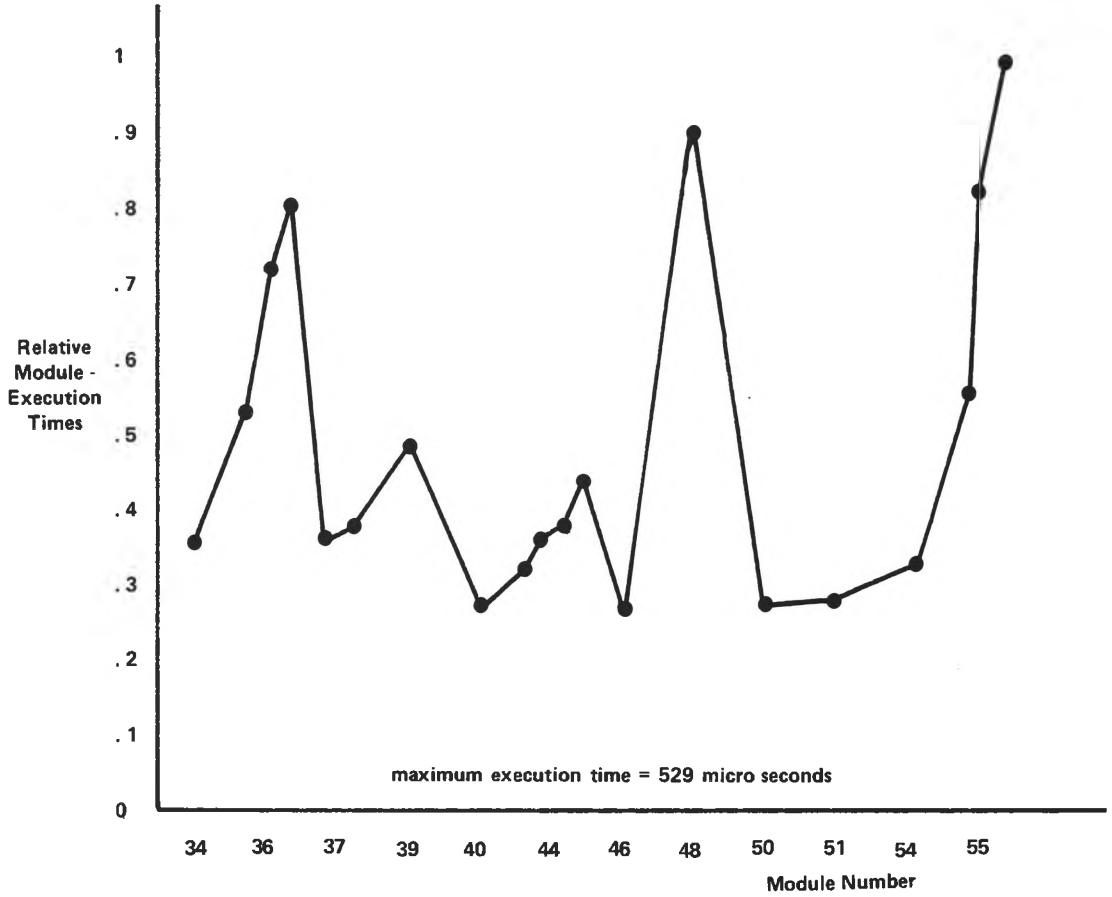


Figure 2.9 Module-Execution-Time Profile for the modules used by Terminal Administrator (entry only - figure 2.4) when executing the paths shown in figure 2.6.

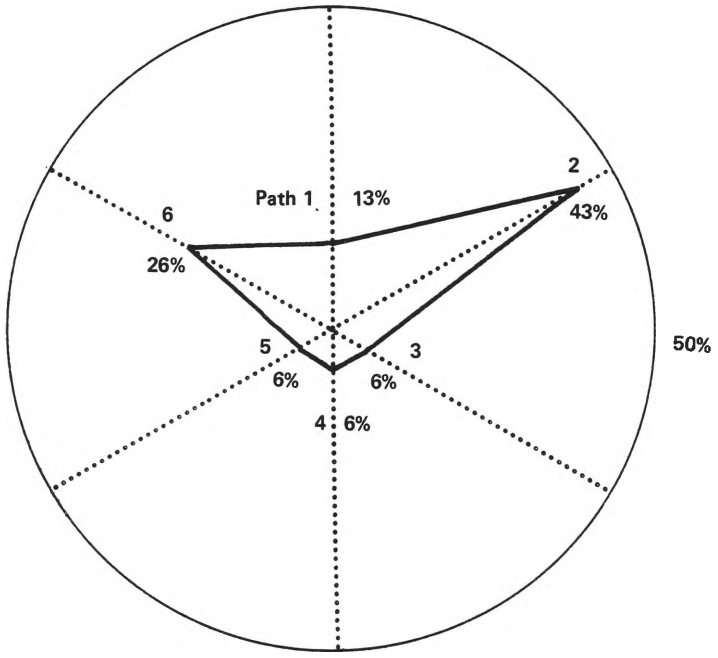


Figure 2.10 Kiviat Graph showing the Utilizations of the Execution Paths shown in figure 2.6.

the set of execution times (T_e) for each path in the set of execution paths (P_e). In both cases the graph is a series of vertical lines (figure 2.5), one for each path. Stimulus information can also be recorded on the spectral lines. The most frequently executed paths, and the paths which take a long time, can be read directly off the spectrum. Thus, areas of poor performance, and execution paths that should be optimised, are pinpointed. The idea of a program having a spectrum, realised by myself as the object spectrum detailed here, was first suggested by my supervisor, Juris Reinfelds, during discussions about data reduction.

- An **execution-path profile** (figure 2.6) can be drawn by plotting the set of execution frequencies (F_e), or the set of path-execution counts (N_c), against the set of execution paths (P_e).

- A **module profile** (figure 2.7) is a histogram of the set of module execution counts (N_m) against the set of modules (O). Again, a normalised version can be drawn by plotting the set of relative module-execution frequencies (F_m). This graph highlights modules with very high usage and modules which are not used at all. Thus, it is a tool for bottle-neck identification.

- A **path-execution-time profile** (figure 2.8) is a plot of the set of object execution-times (T_e), or the set of relative path-execution-times (T_r), for the set of object execution-paths (P_e).

- A **module-execution-time profile** is a plot of the set of relative module-execution-times for the set of modules (figure 2.9).

- **Kiviat Graphs** (Kolence and Kiviat 1973), **Gantt Charts**, and **Utilization Profiles** can be used to display utilization information (figure 2.10).

- Two graphs used to display memory-reference behavior are the **working-set curve** and the **page-fault-rate curve** (page 57 Spirn 1977 - section 3.8).

The execution-path profiles and the execution-time profiles separate out two sets of information that are combined in the object spectrum. Normalised histograms can also be drawn as pie charts. In addition, some of the measurements can be profiled with respect to the value of a stimulus variable, for example work load, job class, data size, disk track, etc.

The measurements also lend themselves to **statistical** methods of data reduction and analysis. For example, one way of reducing the amount of data passed to higher levels is to pass the mean execution-time, and standard deviation, of an object rather than a large set of execution times.

We can construct a data base containing the extent of objects at all levels in the system by extensive measurement of a fully instrumented system. Given such a data base we can hypothesize three **recursive algorithms** for data reduction and analysis.

1. Given a set of stimulus data for an object we can predict the execution path and hence the extent of the other parameters. Thus, by combining the data base with a recursive search algorithm we can produce a powerful simulation model with which we can predict the behaviour of the system under various conditions.

2. Given a measurement of the extent of an object at level n we can determine the extent of the object at lower levels, again using a recursive search, and evaluate what actually happened during the measurement period.

3. Given a graphical interface to the data base, we can walk through an object by interactively selecting the execution path (for example) and seeing what conditions are required to produce different object states.

Validating these hypotheses is a subject for further research, as is the total instrumentation of a system and the construction of an object data base.

2.9. Performance Evaluation

Using the performance measurement formulation (section 2.6), we can now extend our previous definitions of performance (section 2.3).

$$\mathscr{P} = \frac{\mathscr{P}_m}{\mathscr{P}_i} \qquad 2.2$$

$$\text{where } \mathscr{P}_m \subseteq \mathscr{E} \qquad 2.48$$

A set of performance indices ($\mathscr{P}_i$) can be defined, one for each measurement.

$$\mathscr{P}_i = \left\{ \mathscr{P}_{Pe}, \mathscr{P}_{Tb}, \mathscr{P}_{Te}, \mathscr{P}_{Fe}, \mathscr{P}_{Up}, \mathscr{P}_{Um}, \mathscr{P}_{Fm}, \mathscr{P}_{No}, \mathscr{P}_{Nt}, \mathscr{P}_{Tr}, \mathscr{P}_{E\cdot}, \mathscr{P}_{Ts} \right\} \qquad 2.49$$

where the subscripts are the symbol for the respective measurements

However, as mentioned in section 2.2, giving values to these indices is not so easy. In any performance study the evaluator may only wish to examine a subset of these, hence measured performance ($\mathscr{P}_m$) is defined as a subset of the extent. The aim of any modifications to improve performance is to get the measured performance to approach the performance indices.

$$\mathscr{P} \rightarrow 1 \qquad \text{- See Errata point 3.} \qquad 2.50$$

Similarly we can extend our definition of correctness ($\mathscr{C}$).

$$\mathscr{C} = \frac{\mathscr{C}_m}{\mathscr{C}_i} \qquad 2.3$$

$$\mathscr{C} = \frac{\mathscr{T}_{d\,m}}{\mathscr{T}_{d\,i}} \qquad 2.51$$

$$\mathscr{C} = 1 \qquad 2.52$$

for a correct object

Correctness for a correct object is the ratio of the set of measured transformations on the data ($\mathscr{T}_{d\,m}$) and the set of required transformations on the data ($\mathscr{T}_{d\,i}$), and equals one for a correct object.

2.10. Validation of Formulation

There are a number of methods of testing and validating a formulation:

- carrying out experiments to see if the results conform with those predicted by the formulation,
- investigating current practice to see if it fits the formulation, and
- applying the formulation to currently unanswered questions in the field, to see if it provides any insight into those problems.

A hybrid performance-analyser has been designed, and implemented. A partial description of this tool is given in section 6.6, and some measurements at the program-execution level are given in chapter 7. A small computer system, a terminal multiplexer for a local-area network, has been designed, and implemented, with a full set of instrumentation based upon this formulation (section 8.4). The measurements shown in the graphs, included in this chapter, were made on this system using the hybrid performance-analyser.

2.11. Current Measurement Practice

Another method of validation is to compare the formulation to current measurement-practice. One or two examples at each level are discussed briefly. The PRIME system (Ferrari 1973) included two firmware tools for monitoring **microcode-level** activity (section 9.4.1). One measured the utilization of the functional units (modules) of the processor (object). The other measured the flow-of-control (execution path) of the microprograms.

Measurement at the **machine-code** level has changed little since the measurement of Maniac (Herbst et al 1955) programs. Today, the instruction-mix measurement (Gibson 1970) is being used in the optimisation of microcomputer instruction-sets (Fairclough 1982). If the set of instructions is an object then the instruction mix is the set of module-execution frequencies.

Knuth (1971) used two different approaches in the study of the execution of Fortran programs at the high-level-language level: the method of program profiles and the method of

program-status sampling (section 4.4.2). The program profile was a module-execution-frequency table where the object was the program and modules are the Fortran statements. The second method produced a module-utilization histogram which showed how much time was spent in each module rather than how often it executed. Knuth's suggestion that such measurements should impact compiler design have received an unusual twist in the design of the RISC microprocessor (Patterson and Sequin 1982) where module-execution frequencies, and utilizations, for high-level languages were used in the selection of hardware features to optimize instruction-execution times.

Studies of programs at the **block-structure** level have involved the insertion of high-level traces (Reinfelds 1983) and checkpoints (Ferrari 1978a) into programs. A high-level trace involves the insertion of a call to a trace routine, guarded by a boolean variable, at the start of each procedure. The trace routine records the procedure name and some stimulus information (the amount depends upon the level of the trace). From the trace an object execution path can be constructed. Example 9.3 in Ferrari (1978a) demonstrates the checkpointing of a program. The proposed measurements fall into the following classes: execution time, module utilization, module-execution frequency, and execution path.

At the **program level**, benchmarks (see Patterson and Piepho 1982 for an example) of programs on various systems are a very common measurement of execution time. In the design of an Ada debugging tool (Holdsworth 1983), answers to the following questions were considered to be a sufficient set of debugging aids when looking at program crashes:

- Where am I?
- Where have I come from?
- What are the values of the variables?
- How did I get here?

The above questions all pertain to the state (equation 2.47) of the program.

The **task level** is one of the more difficult levels at which to make measurements. Understanding individual processes in an operating system is relatively easy but understanding the complex web of interaction between processes can be a mind-boggling exercise. The instrumentation of Multics (Saltzer and Gintell 1970 - section 8.1) recorded:

- execution time and frequency of execution of supervisor modules,
- the utilization of memory segments and hence module utilization,
- the number of page misses per segment,
- the number of procedure calls,
- queue lengths (stimulus information), and
- the effect of the systems multiprogramming effort on individual users (execution time as a function of stimulus information).

Working set measurements are really measurements at the **system level** and involve the interrelation between the hardware and the operating systems. The object is the set of memory pages and the measurements are profiles of page usage and page misses. Other system level measurements include the measurement of CPU utilization, system-response time, throughput rate, capacity, etc. all of which can be described in terms of appropriate objects.

One area where the formulation may need to be expanded, to take account of the complementary nature of computers and people, is in the measurement of behavioural issues (Miller and Thomas 1977 - section 10.2). However, measurements that have been made of the use of graphics-input devices (English et al 1967), and keyboards (Montgomery 1982), have involved the measurement of response times, which can be considered to be the set of execution times of the object - person plus input device.

The measures described in the formulation of performance measurement can also be used to obtain the extent of parallel objects. Utilization profiles give some clue as to the amount of parallelism. However, to measure the overlap of parallel modules, timing measurements have to

be synchronised to an appropriate time base. Further research is needed in this area (chapter 9).

2.12. Models

Kumar and Davidson (1980) have argued that a hierarchy of performance models, ranging from analytical models to detailed simulation models, is a very useful tool in the design of computer systems. The development and validation of models, both structural and functional, requires the measurement of actual-system values. Application of the formulation to modelling is subject to further research (section 10.1), however, the following are proposed:

- Service times, and visit ratios, for analytical queueing-models (Rose 1978) can be calculated, from data produced using the measures in the formulation, by taking the mean of the execution times of the appropriate modules, and the mean of their frequencies of execution. Multiprogramming level, and job class, can be obtained as stimulus information.
- A simulation model represents the behaviour of a system in the time domain. As there is a conceptual similarity between simulation and measurement, the results obtained by measuring execution time, and execution path, can be used for the development, and validation, of simulation models.

2.13. Corollaries

Finally, we will hypothesise a number of outcomes of the application of this formulation, if it is valid, to performance measurement. Firstly, this formulation provides the general, overall context within which measurement and evaluation can take place (section 6.1), by codifying the field into a unified body of knowledge. Having defined the problems to be studied, the first step in the design of a measurement experiment is to select an appropriate object and divide it into modules. Then, the stimulus variables are defined. Finally, a measurement tool suitable for that level in the hierarchy can be chosen, and the required subset of the standard measurements made. Hence, applying the formulation to actual measurement situations should produce the following results:

- A clear definition of what the parameters to be measured are, at each level in the hierarchy (section 6.1).
- The design of a general-purpose performance-analyser (section 6.4 - technology is no longer a serious limitation).
- A methodology for measuring the extent of objects at each level in the hierarchy (section 6.1).
- A methodology for including performance instrumentation into a system at design time (section 8.3).
- A set of criteria for the partitioning of the monitoring functions between hardware and software, and the cooperation and interaction of these functions, in a hybrid tool (section 6.4).
- A specification detailing the signals needed on the pins of a microprocessor for performance measurement (section 8.3).

All these are discussed in later chapters. Further research, by other researchers, will either validate the formulation of performance measurement, or indicate the need for revision and expansion.

2.14. Conclusion

A computer system is an implementation of a mathematical object, which can be measured. The performance measures can be described mathematically. An object is defined recursively in terms of lower-level objects. A set of measures, which apply at every level of the hierarchy, has been defined. A number of graphical representations of these measures have been demonstrated. The formularizarion of the above measures into a set of equations is a formulation of performance measurement.

This formulation is being validated by:

- designing and executing experiments to test the formulation,
- comparing it to current practice, and
- hypothesising and testing corollaries to the formulation.

Results, so far, indicate a high degree of fit between the formulation and current practice. The measurements illustrated in the figures in this chapter were all made during experiments designed to test the formulation.

The formulation provides a general, overall context within which measurement and evaluation can take place. The purpose of measurement is not to collect numbers, but to gain insight into the actions of the objects under study. The appropriate selection of stimulus information, aided by the use of graphical techniques of data analysis, gives meaning to the actions of the object.

Further research is needed to extend, validate and apply the formulation. A number of research areas, which flow out of the formulation, have been proposed.

3. Other Formulations and Theories

There is nothing new under the sun.

Ecclesiastes 1.9

In the introduction to the previous chapter, we looked briefly at other formulations on which the formulation of performance measurement is based. Now, we will consider a number of other formulations in more depth, particularly their conceptual model of an executing program, to see how they complement, predate, and are subsumed by the formulation of performance measurement. These formulations, and theories, show the rich capacity of the human mind for perceiving the one object from many different perspectives.

3.1. Software Science - Halstead (1977)

Maurice Halstead undertook an empirical study of algorithms to test his hypothesis that the count of operators, and operands, in a program is strongly correlated to the number of bugs discovered in the program. As a result of this study, he developed a set of laws to characterise algorithms. Like many others, he sought a way of analysing the complex problem of software production. Some researchers (Shen et al 1983) have raised serious questions about the underlying theory of software science, while others (Christensen 1981 et al, Fitzsimmons and Love 1978) have produced experimental evidence supporting some of the metrics.

It is generally agreed that the simple measure of counting the number of lines of code is inadequate for predicting programming effort. It is also agreed that either a model of the programming process, based upon a manageable number of major factors, or a scientific theory of algorithm development, is needed.

Halstead proposed an idealised software cycle, similar to the Carnot cycle in thermodynamics, which consisted of four processes: compilation from high-level language to assembly language, optimization of the assembly language, decompilation of the assembly language back

into a higher-level language, and finally expansion of the resultant high-level code back to the original code. As this cannot be done on an actual program without some loss of information, the cycle is considered to be ideal. Measurable properties, operators and operands, were developed so that relationships between points on the cycle could be determined. The machine code concepts of operator: those parts which affect the value or ordering of operands, and operand: the variables or constants, are used to split a program, written in any language, up into tokens. All software science measures are functions of the counts of these tokens. The basic metrics are defined as:

$$\eta_1 = \textit{number of unique operators} \tag{3.1}$$

$$\eta_2 = \textit{number of unique operands} \tag{3.2}$$

$$N_1 = \textit{total occurrences of operators} \tag{3.3}$$

$$N_2 = \textit{total occurrences of operands} \tag{3.4}$$

$$f_{1,j} = \textit{number of occurrences of the jth most frequently used operator} \tag{3.5}$$

$$f_{1,i} = \textit{number of occurrences of the ith most frequently used operand} \tag{3.6}$$

$$\textit{where } j = 1 \,..\, \eta_1 \tag{3.7}$$

$$\textit{and } i = 1 \,..\, \eta_2 \tag{3.8}$$

It will be immediately obvious that the metrics are obtained from a static analysis of the program text in contrast to the performance measurement formulation, developed in the previous chapter, which is based on the dynamics of execution. Hence, at most, we can expect the two hypotheses to complement, but, as they are based upon different axioms, they can neither prove nor disprove each other.

Generally, any symbol or keyword in a program that specifies an algorithmic action is considered to be an operator, while a symbol used to represent data is considered an operand. Most punctuation marks are categorized as operators.

The vocabulary of a program η is:

$$\eta = \eta_1 + \eta_2 \tag{3.9}$$

and the length of the program N is:

$$N = N_1 + N_2 \tag{3.10}$$

and

$$N_1 = \sum_{j=1}^{\eta_1} f_{1,j} \tag{3.11}$$

$$N_2 = \sum_{i=1}^{\eta_2} f_{1,i} \tag{3.12}$$

The length N is obtained by observation in contrast to a second length measure $\hat{N}$ which is calculated:

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \tag{3.13}$$

An analysis of the first dozen algorithms published in the Algorithms section of the Communications of the Association for Computing Machinery showed a very close correlation between measured and calculated length. This result, however, is based upon programs written by experts. Can the same be said of programs written by average programmers?

Additional metrics are defined using these basic terms. The volume V of a program is the actual size of a program in a computer in bits, if a uniform binary encoding for vocabulary is used.

$$V = N * \log_2 \eta \tag{3.14}$$
$$= \textit{length} * \textit{vocabulary}$$

The potential volume $\hat{V}$ of a program is the volume not of the optimum solution for the problem, but of the smallest piece of code required to invoke a solution to the problem, for example a procedure call with operands as parameters. This makes LOGO one of the highest level programming languages, because of its ability to define procedures as new words in the language. Thus, volume becomes a measure of program level L rather than a measure of algorithm quality (optimality).

$$L = \frac{\hat{V}}{V} \tag{3.15}$$

The value of L ranges from zero up to one, with $L = 1$ representing a program written at the highest possible level (i.e. minimum volume). As the volume of a program increases the program level decreases and the difficulty D increases.

$$D = \frac{1}{L} \qquad 3.16$$

Thus, the use of high-level language constructs should reduce the difficulty of programming. The effort E required to implement a computer program increases as the size of the program increases.

$$E = \frac{V}{L} = D * V \qquad 3.17$$

The time taken to implement a program $\mathscr{T}$ is a function of the programming effort and the time required by the human brain to perform the most elementary discrimination.

$$\mathscr{T} = \frac{E}{S} \qquad 3.18$$

where $5 \leq S \leq 20$ *discriminations per second*

Halstead's intuitive model, upon which this work is based, appears to be: programs are produced by programmers working through a process of mental manipulation of the unique operators and operands. Thus, volume can also be interpreted as the number of mental comparisons needed to write a program and programming effort is measured in terms of elementary mental discriminations. Coulter (1983) has questioned the validity of this model on the following psychological grounds. Short-term memory properties have been incorporated into models involving other memory stages. Studies on human memory searches do not support a fundamental software science conjecture: that humans use a binary search - implying that we store things in ordered lists. Results concerning psychological time appear to be misused in software science; consequently, those theories were applied to the wrong stages of memory. Following this summary of his objections, Coulter goes on to discuss the objections and their implications in detail.

In his book, Halstead (1977) also looks at the ratio of operators to operands, program intelligence content, and program purity. The criticisms of software science are discussed in detail by Shen et al (1983). Two theoretical problems are raised: firstly, no method of classifying operators and operands in high-level language programs is provided, and secondly, the

weakness of some assumptions. A large amount of empirical evidence suggests that software science is the best measure of programming effort we now have.

As mentioned previously, software science does not have any comment to make up on the formulation developed in the previous chapter. It is based upon a different set of axioms, and seeks to answer a different set of questions. It has been included in this text for completeness.

3.2. Software Physics - Kolence (1972)

While many researchers have taken up Halstead's work, few have done independent tests of Kolence's formulation (Morris 1976, Prichard 1976, Febish 1981), yet it bears greater relevance to performance measurement. Software physics deals with computer sizing and workloads, and has been used as a basis for the development of software performance monitors. Kenneth Kolence postulated a software physics based upon the same fundamental concepts as natural physics. Then, he developed performance measures based upon these concepts, and sought to verify them empirically. Kolence saw that without a unified formulation the computer-performance-measurement field would continue to be an aggregation of measurement methods characterised by a state of confusion about how to interpret the data.

Fundamental to Software physics is the idea that a piece of software is composed of a sequence of operations. A **software unit** is a set of transformations and other relationships (e.g. positional, etc.) over data, container and symbol structures. Kolence defines the terms data, symbol structure and container by example and does not give precise definitions. This is partly due to the fact that the definition of a software unit is essentially hierarchic, and hence examples at levels in the hierarchy are easier to understand than a general definition. **Data** is the information being processed. **Symbol structure** is the labels by which the data is referenced and changed (i.e. the code). A **container** is the thing which holds the data, for example a register at the machine-code level, or a stack at a higher level.

The definition of an object given in section 2.5 is essentially identical to a software unit. It has the same hierarchical nature, and consequently, the same claim to a set of relationships

which hold across the hierarchy. Kolence expressed his ideas in English, not in Mathematics.

The transformations and other relationships mentioned in the definition of a software unit have a domain: those containers from which the contents are transferred, and a range: those containers which receive the domain contents. The domain is effectively the set of inputs to the object and the range is the set of outputs.

Those properties of a software unit which are dependent on both structure and relationship in the hierarchy are considered to be **level-dependent**, and those properties which are independent of relationships in a hierarchy, and will therefore add up to the total value of the system, are considered to be **level-invariant**. Kolence then hypothesises the existence of conservation laws for all level-invariant properties of software units.

The basic concepts of software physics are the software force, work, and energy of a software unit. **Software energy** is the capacity to cause a change in the state of a software unit. **Software work** is the actual change in energy state of a software unit affected by some transformation. One unit of software work is performed on a storage medium when one byte of that medium is altered. **Software power** is the software work performed per unit-time.

Software force F is defined by analogy with quantum physics as:

$$F = \frac{1}{C} * \frac{\Delta E}{\Delta t} \qquad 3.19$$

where ΔE *is the change in energy in ergs*

Δt *is the time taken*

and C *is the velocity of light*

The direction of the force is from the domain of the transformation to the range. This definition raises the question: Is Kolence stretching the idea of a correspondence between natural physics and software physics a bit too far? After all, what does the speed of light have to do with the execution of a program?

Although these concepts are necessary to demonstrate the linkage between natural physics and software physics, the direct measurement of work and force are rather difficult. Also, the

software energy available in a system is quite complicated to calculate. Thus, operational definitions expressed in more measurable terms are needed to form a usable formulation for performance measurement.

Units of work are defined directly in terms of the changes in the contents of known containers. For example, we can measure the work (W_o) done by an I/O device, which transfers data in n byte blocks, by simply determining the number of containers (blocks) transferred.

$$W_o = \sum \textit{blocks transferred by each I/O operation} \qquad 3.20$$

Also, the work (W_c) done by a central processing unit is the number of memory accesses used by the central processing element, and thus is a function of the instruction mix (k) for a given program workload. Information theoretic work (W) done by a work load is:

$$W = kW_c + W_o \qquad 3.21$$

where $k = 1$ *for most uses within an installation*

A relationship (equation 3.22), which holds at all levels of the machine and software hierarchy, can be established between the work (W) performed by the machine and the work (W_{Pi}) performed by some software unit (P_i).

$$\begin{aligned} W &= \sum W_{Pi} = k\sum W_{cPi} + \sum W_{Pi} \qquad 3.22 \\ &= kW_c + W_o \end{aligned}$$

Thus, software work is conserved, and is level invariant with respect to both job mix and equipment hierarchies. The software energy available is the total number of memory accesses which can be used, given one hundred per cent cpu utilization and maximum I/O transfer rates. Software power (P), or rate of software work, is the work done per unit time.

$$P = \frac{W}{\Delta t} = \frac{kW_c}{\Delta t} + \frac{W_o}{\Delta t} = kP_c + P_o \qquad 3.33$$

The overall power usage of a machine configuration can be measured by measuring the number of memory accesses used and the time required to perform this work. Thus, power usage is proportional to cpu utilization - one of the standard performance measures. On the other hand, the I/O power usage is not proportional to I/O device busy time, due to the variable rate of

data transmission, and transmission time includes setup times (e.g. seek time) when no transmission can occur. Consequently, I/O power is seen to be level dependent.

The maximum software energy that could be used is the product of the maximum power usage that can occur over any period of time and the length of the time period. The software energy available for use by software units is partially dependent upon the machine configuration, the characteristics of the work load, and the operation of I/O devices.

Kolence claims that by empirically determining common workload properties we should be able to characterize different types of computers, and I/O devices, in terms of their effectiveness to process various classes of workload. Thus, we have a scientific method for computer system selection. Software Physics deals only with capacity management: how to keep the computing equipment matched to the changing workload in a particular system. Kolence gives some experimental data to support this claim, but others seem not to have been convinced as they have not taken up this methodology.

Kolence started with a definition of a software unit very similar to the definition of an object given in chapter 2, but his world view led him in a different direction to the formulation developed in chapter 2. The idea of a software physics deeply related to natural physics has not been proven, however the measures produced (work, energy, force, and power), provide a conceptual framework for thinking about performance.

Having got the numbers what do they mean? Kolence (1975) replaces a number of traditional measures; for example number of lines printed, cpu seconds, number of I/O operations; with measures of work and work vectors. While no one questions the importance of the concept of work in physics, they still measure the work done by a tip truck in cubic metres of earth moved, not in ergs. The cpu power used by an object is proportional to the module utilization, when the cpu is one of the modules in that object. In the case of the power and the utilization of an I/O device the situation is more complex. The question - Which measure, power or utilization, is more meaningful? - requires careful consideration.

3.3. Program Performance Indices - Ferrari (1978)

Domenico Ferrari (1978a) devotes a whole chapter (chapter 9) in his book to the discussion of program performance. Knuth (1968) has defined a program as: *An expression of a computational method in a computer language.* From the performance viewpoint, a program can be considered to be a black box which takes input data and transforms it to output data. The time taken for a correct program to execute is a measure of the degree of difficulty of the transformation. When comparing two programs which execute the same transformations, execution time becomes a performance measure. A second, related index, is the price we have to pay for the resources used by the program.

The basic question of program performance analysis is: How does a program spend its execution time. If a program is considered to be a set (S) of disjointed states (S_1 .. S_n) then the behaviour of a program can be described as the sequence of states, visited during execution, and by the time spent in each state.

$$S = (S_1 .. S_n) \tag{3.34}$$

$$Sequence = \left\{ i_j(t_j) \right\} \quad \left[j = 1,2,\cdots; i_j \in (1 .. n) \right] \tag{3.35}$$

where t_j is the duration of the jth interval of execution which is spent in state S_{ij}.
and i is the state number

If we define

$$t_j(k) = t_j \text{ when } k = i_j \tag{3.36}$$

$$t_j(k) = 0 \text{ when } k \neq i_j \tag{3.37}$$

$$\text{for all } j \text{ and for } k = 1..n$$

then the total time spent in state S_i during the execution of the program is:

$$t(i) = \sum_{j=1}^{r} t_j(i) \tag{3.38}$$

where r is the number of intervals into which the execution time has been subdivided.

The execution time (t) is

$$t = \sum_{i=1}^{n} t(i) \tag{3.39}$$

This is clearly a subset of the formulation developed in chapter 2, in which object execution-time and module (if a state is considered to be a module) execution-time are defined. The representation is more complex, but the concepts are the same. The decomposition of the object execution-time into module execution-times, and associated module execution-paths (the sequence of states visited during execution), enables the characterisation of program performance in terms of resource demand patterns, if modules correspond to various hardware resources. This characterisation is very close to that upon which charges for program execution are normally based.

Other types of decompositions, leading to different performance indices, are also possible. For example, a program may be divided into blocks. At any instant during the measuring period, the program will be accessing one of these blocks. Referencing a block, or a distinct subset of blocks, may be viewed as a state of the program, and the execution time may be decomposed into the intervals spent in each state. The sequence of states (execution path) defined by the blocks is a description of the referencing pattern produced by the program when processing a specific set of input data.

Ferrari raises the possibility of other decompositions, for example a program trace, and suggests that practically all the questions regarding the behaviour of a program during execution can be answered if the trace and listing of a program are known. He then illustrates this technique (example 9.1 Ferrari 1978a) by decomposing a Fortran program into blocks, first at the subprogram level, and then at the instruction level. In each instance execution paths and execution times for the blocks are measured and discussed. If the frequency counts for each statement are known then a program profile can be drawn. Ferrari then goes on to discuss the use of a program's address-trace, or reference string, in studying the page allocation strategy in a virtual-memory system.

The reader will have already observed the similarity between Ferrari's work and the formulation developed in chapter 2. In fact, the performance measurement formulation is a generalisation and expansion of Ferrari's program performance indices, and embodies much of it.

Kolence's idea of a software unit has been used in the generalisation and a full set of decompositions have been defined. Ferrari's viewpoint is slightly different: that of considering a sequence of states compared to that of considering a sequence of modules which are initiated by state changes, but the result is the same. Thus, Ferrari's chapter (1978a,chapter 9) on *The Evaluation of Program Performance* is a rich source of ideas, both for applying the formulation to measurement, and for evaluating the results of those measurements.

3.4. Measurement Concepts - Svobodova (1976a)

Liba Svobodova's book on Performance Measurement includes a small section (chapter 6.1) on measurement concepts. The measurement problem is to determine:

- what information is pertinent to a specific measurement objective,
- where such information can be found, and
- how it can be extracted and recorded.

Svobodova's model of a system is that of a sequence of changes of system state, where information about system state is contained in the system's memories. A change in system state marks either the beginning or the end of a period of activity (or inactivity) of a system component (hardware, software or process). Since several components (processes) can be active simultaneously, a change in the system state is a change in the level of system activity. A change in system state is called an event. Events can be both hardware and software related.

The system state can be described by a vector composed of binary elements representing the states (O or I) of individual memory elements. An activity a_k can then be represented by a logical function that has the value of 1 on a subset x_k of the set of all possible states x, $x_k \subset x$. An initiation event e_k occurs when an activity a_k begins, causing the system to change from state $x_{old} \neg \in x_k$ to state $x_{new} \in x_k$. A termination event $\neg e_k$ occurs when an activity a_k terminates, causing the system to change from state $x_{old} \in x_k$ to state $x_{new} \neg \in x_k$.

Using this model of system behaviour, measurements can be divided into four categories according to the type of information recorded about the activity. For a measurement period $t_o \overset{\bullet\bullet}{\times} t$ the categories are:

- An event trace is a sequential record of all initiation and termination events during the measurement period. An activity can be completely described by a sequence of pairs $\left(t_{ki}, T_{ki}\right)$, where t_{ki} is the time of the ith occurrence of this activity and T_{ki} is the corresponding duration of this activity. The result of obtaining this information from the event trace is a time stamped record of all the invocations of the activity, indicating when the activity started and its duration on each occasion.

  No allowance for the pausing and resumption of an activity appears to be made, unless the resumed activity is considered to be a separate activity. The problem with the latter is that the pause can be caused by an external agent, for example an interrupt, which has no relevance to the current activity, and hence the place where the activity pauses is arbitrary. Interaction between activities has not been considered by Svobodova, but it could be studied by drawing the event trace as an activity flow-graph.

- Relative activity r_k is the ratio of the total time of the activity a_k and the total elapsed time.

$$r_k = \frac{1}{t - t_o} \int_{t_o}^{t} a_k(\tau)\, d\tau \qquad 3.40$$

  where $t - t_o > 0$,
  $a_k = 1$ if $x(\tau) \in x_k$,
  and $a_k = 0$ *otherwise.*

- Event frequency (c_k) is the number of times an activity is initiated during the measurement period i.e. the same idea as module-execution- frequencies.

$$c_k = \frac{1}{t - t_o} \sum_{t_n} e_k(\tau) \qquad 3.41$$

  where $t \geq t_n \geq t_o$,
  $e_k = 1$ for $\tau = t_n$,

$e_k = O$ *otherwise*,

and t_n *is the number of occurrences of* e_k

- The distribution of activity intervals ($f_{kn}(T)$) is the distribution of the duration times (T) of an activity (a_k) at the time of the nth termination of the activity, i.e. the same idea as an execution-time histogram.

$$f_{kn}(T) = \frac{1}{n} \sum_{i=1}^{n} g\left[T, T_{ki}\right] \qquad 3.42$$

where $g(T,d) = 1$ for $T = d$

and $g(T,d) = 0$ *otherwise.*

The measurement categories presented by Svobodova all have equivalents in the formulation developed in chapter 2, and in fact are a subset of same, although the mathematical description reflects a slightly different perspective on program execution.

Ferrari, Svobodova, and McKerrow are using the same model of program execution but are looking at it from different perspectives. In each case, the concepts of time sequence of events, state change (event), and activity between state changes (modules) are present. Ferrari emphasises the time sequence of states and pays little attention to the activity between states, on the assumption that significant activity will be reflected by the occurrence of a new state. Svobodova emphasises the occurrence of events (state changes) as these indicate the start and end of activities (modules), on the assumption that we are interested in how often each activity occurs and for how long. I am emphasising the activities (modules); using the events to detect initiation, pausing resumption, and termination of modules; and using the state (stimulus information) to give meaning to the action of the object. My perspective is, we want to know why execution has occurred the way it has, as well as what execution has occurred, how long the execution took, and the order in which the modules were executed. Thus, I have attempted to produce a more comprehensive measure of system activity.

3.5. Monitoring Program Execution - Based Concepts - Plattner and Nievergelt (1981)

Bernhard Plattner and Jurg Nievergelt were interested in dynamic program analysis, particularly

as a debugging tool for use during software development. They developed a set of basic concepts of program execution which they used as the basis of a series of execution-monitoring experiments. The program under test was compiled, using a modified compiler, and run on a target computer. During the execution of the target process, a hardware break-point-device, which has been added to the target processor, sends state information through a first-in-first-out queue to a monitoring processor. The monitoring processor uses the state information from the queue together with an extensive description of the target process, which was produced by the modified compiler, to model precisely the execution of the target process. This monitor works in real time, does not impact the target process, and uses the queue to solve synchronisation problems. However, the monitor appears to be suitable for the study of programs in isolation only.

Plattner and Nievergelt think of a process as a trajectory of a point moving through space. The monitoring activities of interest are defined as: requesting notification when the trajectory enters some prespecified region, halting the motion of the point, and restarting a new trajectory at some other point in space. The structure of this state space S(P) is completely defined by the program P in execution, where P is written in a programming language PL , and the semantics of the programming language. A point s in S(P) is a potential state of some execution of P: it corresponds to an assignment of values to all variables of P and a specification of a point of control PC. By executing P starting in an initial state s0, a process denoted by p(P,s0) = s0 .. sn, will be created. In this context, the PC is not simply the hardware program counter, but includes all procedure-calling chains of the target process. A process state consists of a control component: the PC, and a data component: input data and internal variables. Thus, the trajectory is defined as the sequence of states through which the program moves as it executes.

The monitoring process requires that the entire state space be accessible i.e. the monitoring process should have a complete record of all the states of the process as well as the states through which the trajectory passes. Monitoring is achieved by means of predicates that assign

a truth value to a target process, and actions that modify the target process (they wanted the ability to stop the target process and restart it under different conditions - a useful facility when debugging programs). Predicates fall into two classes: process predicates which answer questions relating to the process, for example how often are two statements executed in sequence; and state predicates, for example let me know when the trajectory crosses the boundary from one region into another.

Plattner and Nievergelt also discuss the complexity of the structure of the state space with respect to programming-language data-structures, and to dynamic memory-allocation. The state space of a recursive tree-traversal-program is discussed to illustrate some of the details involved in defining the notions of process state and state space.

These concepts are based on a description of program execution very similar to Ferrari's, but the underlying conceptual model is different. This model is useful for studying processes in isolation, but is restrictive when studying interaction between processes and processors. Is each process a separate trajectory with a separate point, or does one point traverse all trajectories, or do we have several spaces each with their own points and trajectories? In each of these cases, how do you describe the jump from one trajectory to another? Thus, their formulation is not as general as the one developed in chapter 2, but it does provide an interesting perspective on a subsection of that formulation: that debugging programs and performance evaluation are differences of application of formulation and tools, not conceptual differences in either formulation or tools.

3.6. A sequential Program Model - Franta et al (1982)

Franta et al (1982) worked on the development of a distributed system testbed at the Honeywell Corporate Computer Sciences Centre. Instrumentation of computer systems is concerned with the observation and control of events, specifically those that constitute or precipitate changes of computer system state. The notion of an event is formalised using models of both centralised and distributed systems.

A variable or data object x is an entity with a name "x" that can take on any value $V(x)$ of a certain defined set of values, and upon which any of a defined set of operations can be performed. The set of values together with the group of operations is called the data type of x. The state of x is its value $V(x)$. Given a set X of n objects $[x_1 .. x_n]$ where x_i is of type T_i, the current state q of X is given by a vector of values of the objects.

$$q(x) = <V(x_1) .. V(x_n)> \qquad 3.43$$

The state space S(X) is the set of all possible such vectors.

A single terminating sequential program B defined over a set of objects X effects a state transition on X in that it is invoked with the objects in one state q(X), and terminates with the objects in another state $q^1(X)$. Such a program may be modelled as a binary relation M on S(X); M is a collection of ordered pairs of states. The interpretation of this relation is that the pair $<q(X), q^1(X)>$ is an element of M when B is guaranteed to halt when invoked from state q(X), and when $q^1(X)$ is one of the states in which B can halt when invoked from q(X). From this interpretation it follows that the domain of M (i.e. the set of states that can be first components of pairs of M) is exactly the set of initial states from which termination of ~~of~~ B is guaranteed.

A particular execution of the program B over X defines a state sequence.

$$s(X,B) = q_o(X) .. q_n(X) \qquad 3.44$$

where $q_o(X)$ is the state at invocation of B. If $q_o(X)$ is an element of the domain of M, then s(X,B) is finite; and if $q_n(X)$ is the last state in s(X,B), then $<q_o(X), q_n(X)>$ is an element of M.

There is a state transition sequence

$$h(X,B) = t_1(X) .. t_m(X) \qquad 3.45$$

associated with the state sequence $s(X,B)$, where the transition $t_i(X)$ is the ordered pair of states $<q_{i-1}(X), q_i(X)>$. These transitions are referred to as events, and the event sequence is termed the history of the program. The set H(X,B) of all possible history sequences

constitutes the set of possible histories of program B. For a deterministic sequential program B, there is a unique history $h(X,B)$ for each pair of initial and final states $<q_o(X), q_n(X)>$.

This is a perfectly valid conceptual model, albeit rather complex, of program execution based upon the data flow concept, i.e. events are defined in terms of what happens to the data. Once again, an event trace is produced, but the event trace lacks timing information. Also the concept of execution path is missing. Thus, as it is, this model is inadequate for performance measurement.

3.7. A Measure of Computational Work - Hellerman (1972)

Hellerman (1972), Constantine (1968), and Rozwadowski (1973) have attempted to define a work unit based on information theory. Hellerman measures the computational work of a process in terms of the information in a memory for a table-lookup implementation of the process.

Approaches to the problem of defining a measure, involve two steps:

- expressing the process to be measured in some canonical implementation, and
- taking some quantitative attribute of that implementation as the work of the process.

Thus, the problem becomes - can we find a canonical formalism, where one attribute is a measure of work, with which we can implement the universal set of processes.

A number of canonical forms have been suggested: logic based on NAND gates, Turing machines, and formal descriptions in terms of inputs and outputs; but in each case examples can be found where the proposed quantitative attribute fails as a measure of work. Hellerman (1972) proposes a measure that is a function of the number of inputs of the process and of how these inputs are distributed among the outputs. The canonical form of a process is its table lookup implementation, and the quantitative attribute is the quantity of information in memory.

Let $f: X \rightarrow Y$ be a process defined on a finite number |X| of inputs. The domain X may be partitioned into n domain classes X_i, each comprising all points in the inverse image of some point in the range Y. The work of f is then

$$w(f) = \sum_{i=1}^{n} |X_i| \log_2 \left(\frac{|X|}{|X_i|} \right) \qquad 3.46$$

where $|X|$ = *number of possible inputs*

$|X_i|$ = *number of inputs which can produce output* Y_i

n = *number of possible outputs*

Since computational work is measured in terms of information, the unit of work (a wit) is the same as the unit of information (a bit). Computational power (wat) is measured in terms of wits per second.

This contrasts to Kolence's measure of work, where a unit of work is a change in the state of a bit rather than the fact that information is stored in the bit. Thus, this definition of work does not appear to take into account the number of actions performed upon the input to produce an output.

3.8. Program Behavior: Models and Measurements - Spirn (1977)

The development of virtual-memory systems gave impetus to the modelling of programs, as a separate, but parallel, endevour to the modelling of systems. Jeffrey Spirn (1977) attempts to draw together the concepts of program behavior into a survey of the field. He argues that programs must also be modeled for performance studies of systems. A program does many things which might be of interest to model: it references memory, issues input/output requests, generates various kinds of interrupts, acquires and releases resources, communicates with other programs, interacts with the user, etc. In his book, Sprin concentrates on memory reference behavior, primarily because little is known about other aspects of program behavior.

Central to his models of program behavior is the phenomenon of locality. The principal of **extrinsic locality** is: a good predictor of a programs future behavior is its immediate past behavior. In contrast, the principle of **intrinsic locality** is: expected future behavior will only agree with past behavior while the program is in a given state. Thus, an intrinsic-locality model consists of a state-transition mechanism, and a characterisation of the behavior within each state. Extrinsic-locality models are ignorant of internal state-transitions. Thus, extrinsic locality

is specified by an external measurement, where as intrinsic locality is specified by a state.

In the study of memory reference behavior, extrinsic locality denotes the **set of addresses** (or, at a higher level, pages) referenced recently by a task. Intrinsic locality denotes the set of addresses likely to be referenced while the task remains in its current state.

A **program** is a sequence of instructions as written by a programmer. When discussing program behavior we really mean **programs in execution** on a processor. Interruptions to the program, for example process switching, are ignored by postulating a **virtual-time** clock, which runs only while the given program is executing. A program in execution in virtual time is called a **task**.

The memory references of a task are the set r(1) .. r(k), where $r(t) \in N$ is the page referenced at virtual time t and $N = 0 .. n-1$ is the set of pages belonging to the task. A sequence of references r(t) .. r(t + k) in some virtual-time interval is called a **reference string**. S(t) is the set of pages in main memory just after the reference at time t. For all t we have $S(t) \subseteq N$, and $|S(t)| \leq m(t)$, where m(t) is the memory allocation of the task at time t, and the set of page frames in main memory allocated to the task M(t) = 0 .. m(t)-1. The set of pages currently "in use" by the task is the locality set L(t), where $L(t) \subseteq N$ and $|L(t)| \leq n$.

The working set at time t, W(t,T), consists of the set of distinct pages in r(r-T + 1) .. r(t). The working-set size (number of pages) of W(t,T) is w(t,T), and w(T) is the virtual-time-average value of w(t,T) over a specified time interval. The **working-set curve** is a plot of w(T) against T for a given task, where T is the time taken for a task to complete one reference to memory (average of memory references with and without page faults).

In his book, Spirn develops several models of program behavior: a working set model, intrinsic locality models, stack models, an independent reference model, etc. These models are used to study paging and paging algorithms. Measurement for, and validation of, these models is also discussed.

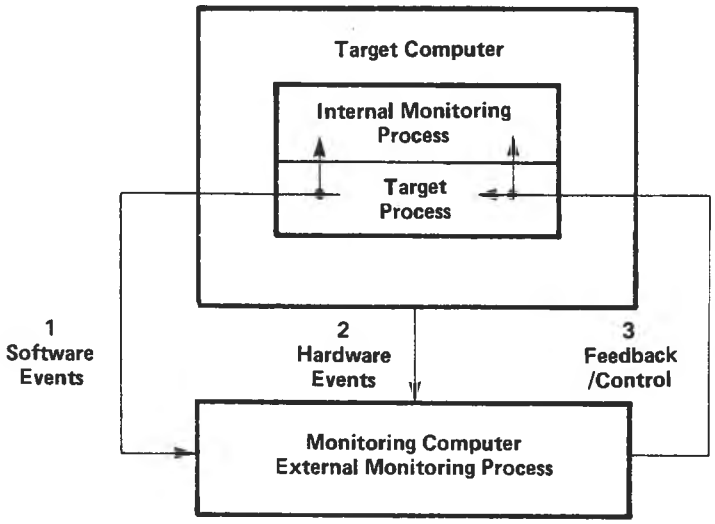


Figure 4.1 Information Paths Between a Target Process and a Monitoring Process

4. Measurement Tools and Techniques

Two approaches, known as the stimulus approach and the analytical approach (Svobodova 1976a), to performance measurement are found in practice. In the stimulus approach, the system is treated as a black box, with a limited number of known functions. Simulated inputs (stimulus) are supplied to the black box and the response of the black box (outputs) is measured. A benchmark is a record of the systems response to a controlled work load.

In the analytical approach, the system is separated into parts for detailed measurement of internal behaviour. At times the two approaches are combined. Both involve measurement, and both can be used at any level in the object hierarchy.

Three data transfer paths exist between a target process and a monitoring process (figure 4.1):

1. Information obtained using software tools is either passed to an internal monitoring process or to an external monitoring process via hardware probes. The latter is a hybrid tool.

2. Hardware event information is collected with hardware probes. This information can be recorded, or transformed using counters and comparators (a hardware tool).

3. Information can be fed back, from a monitoring process to a target process, for use in the adaptive control of the operating system. Control information is used to control measurement experiments.

The data paths used by a monitoring tool to collect information, and whether the monitoring process is internal to the target computer or in a separate monitoring computer, classifies the tool into one of the three major categories: Software, Hardware or Hybrid. Firmware tools are often considered to be a separate classification, but they generally form part of a tool in one of the other categories.

4.1. Measurement Tool Modules

A measurement tool can be divided into four conceptual sections (figure 1.4). The **sensor** section is the interface between the target process and the measurement tool. It is the front end of the measurement tool. The sensor detects events of interest and measures the magnitude of the quantities being monitored. Sensors are often referred to as probes. A software probe is usually a sub-program, inserted into the target process, and a hardware probe is usually a set of connectors, terminated to the back plane or special test points.

Sensors operate in one of two modes: internally driven or externally driven, determined by how the action of sensing is initiated. Externally driven tools usually sample system state in response to the occurrence of an event outside the target system, for example at the end of time periods measured by a clock. Data collected in this way has to be analysed statistically to produce meaningful results. Internally driven tools usually detect events occurring inside the target system, for example a procedure call or an interrupt vector. Thus, the data they collect is synchronised to the internal operation of the target process.

Data reduction occurs in the **transformer** section. Typically, the sensor produces a continuous stream of event descriptors and stimulus information. Only a subset of this observable information is of interest in any one measurement experiment. The transformer selects, and often massages, the subset of measured data to produce the set of measurements relevant to the experiment. Comparators are used to select events of interest from the event stream. These selected events are either recorded or used to trigger the recording of other information, often by mapping the event descriptor to a structure of data pointers. Counters are used to count events. Counter outputs are recorded, and can also be used to trigger the recording of other information. Time stamps can be added to event records and stimulus records.

Data collected by the probes is either in the form of single bits (flags) or in functional groups of parallel signals (words). The operations performed by the transformer section on this data fall into the following categories:

- Data can be stored without change.
- Data can be masked to remove unwanted bits before storage.
- Data can be compared to reference values, and then flags; representing: equal to, less than, greater than, within range, outside range; are stored.
- Data can be logically manipulated by a function generator, for example two signals ANDed, before storage.
- Sequences of data patterns can be detected and stored.
- Sequences of data patterns can be detected and used to initiate storage of selected data sets which occur either before, during or after the sequence.
- Successive data inputs can be compared and the results stored.
- All of the above can be counted, i.e. counting the occurrence of specific data patterns.
- The time period between the occurrence of specific data patterns, in all of the above, can be measured.
- Some of the above can be combined to produce more complex reduction schemas.

The resultant set of information (event trace and stimulus information) is stored in a data base, for example the object record described in chapter 6. Small quantities of data are held in the tools memory and larger quantities are held on back-up store. The creation of this data base is an important function of the transformer.

The **analyser** section processes the data stored in the data base to produce the final output of the experiment, for example tables and graphs. A set of suitable displays was described in chapter 2. These outputs, plus the data base information are displayed by the **indicator** section. The analysis to be performed upon the data is determined by the hypothesis the experiment was designed to test.

Analysis often takes place at a later time using recordings, on tape, of several experiments.

However, there are significant advantages in being able to analyse the data, in real-time, as the experiment progresses (Fuller et al 1973):

- Analysis may indicate a flaw in the measurement specification, or inadequacies in the filtering done by the transformer, enabling these to be modified, thus, curtailing useless measurements.
- Analysis may indicate an error in the hypothesis, confirm the hypothesis, or indicate the need for further experiments. The experiment can be modified and re-run on the basis of the analysis. Thus, real-time analysis will speed up the evaluation process.
- Data can be used to interactively test and debug programs.
- Analysis results can be used to dynamically tune the operating system.

Thus, real-time analysis gives the analysist greater control over the experiment, and reduces evaluation time (Aschenbrenner et al 1971). Hence, the trend is toward on line analysis of measured data, either during or immediately after the experiment, with graphical displays of relevant information.

4.2. Measurement Tool Characteristics

Any measurement instrument can be described in terms of a set of characteristics typical of that class of tool. A set of characteristics (Ferrari 1978a) has been developed for performance monitors. Introducing a tool to a system can impact upon the operation of the system. **Interference** can occur in a number of ways:

- Incorrect connection of hardware probes can degrade signals causing apparent hardware faults.
- Software tools use memory space, reducing the space available to the target process.
- The execution of a software tool uses system resources, causing a degradation in the actual performance and introducing inaccuracies into the measurements.

- Improperly designed software probes can change the operation of the target process, introducing logical errors into the target process.

Two measurements are used to discuss the **accuracy** of a tool. **Precision** refers to the number of digits available to represent data. **Resolution** refers to the maximum frequency at which events can be detected and correctly recorded.

The **scope** of a tool categorises the classes of events which the tool can detect. For example, a time counter used to count CPU time, when the CPU busy signal is set, has a very limited scope. On the other hand, a logic-state analyser can be used in a variety of situations, and thus, it has a wide scope.

The **pre-reduction capabilities** of the tool determine the type and size of experiments for which it can be used. Measurement can be limited by the number of comparators, the number of counters, the size of the plugboard, the available logic functions in the transformer, the number of signal connections, the size of the data storage memory, and the availability of signals for event detection. Lack of suitable event detection, signal recording, and pre-reduction facilities results in loss of information.

Ease of Use has a big impact upon the acceptance of a tool by analysts. Important considerations include: quality of the documentation, interactive setup of the transformer, ability to define events of interest, degree of difficulty of probe insertion (ease of installation), ability to activate and deactivate the recording of events, methods of accessing the data base, and the power of the analysis tools.

Compatibility refers to the matching of signal levels etc. at the interface between the target system and the monitoring system. Electrical voltage levels must be the same and hardware probes must not load signals. Software probes must not violate system protection mechanisms or interfere with the operation of the operating system.

As with other measurement tools, you get the features you are willing to pay for. The **cost** of purchase, installation, usage, expansion and maintenance of a tool should be compared

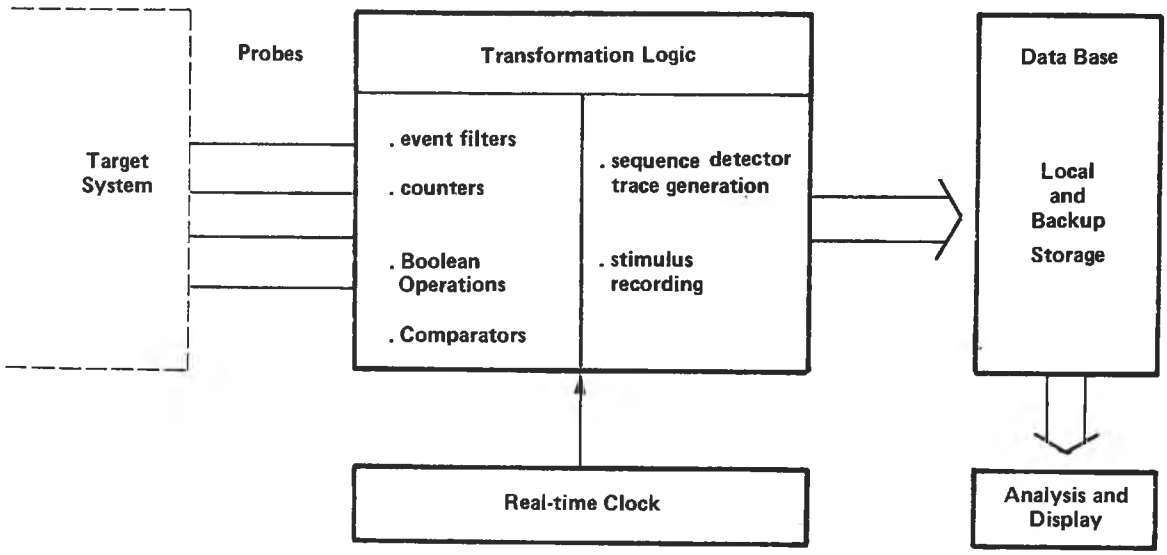


Figure 4.2 Block Diagram of a Hardware Monitor

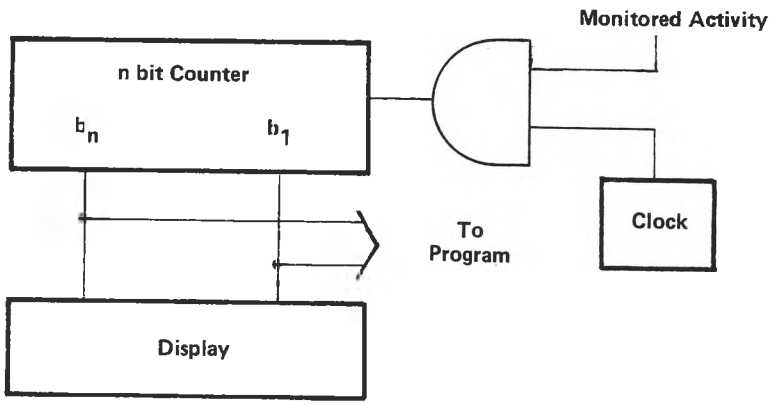


Figure 4.3 Timing Meter

to the expected cost savings due to performance improvement.

When installing a tool into a system three general problems have to be addressed.

1. What impact will the introduction of the tool have on the system?

2. Is a suitable clock available, or does an external clock have to be added?

3. What can be measured with the tool?

Having discussed the properties of measurement tools and techniques in general terms, in the next sections we will look at specific tools, and how they have been used.

4.3. Hardware Tools and Techniques

A hardware tool consists of additional hardware added to the target system to collect signals of interest, and external transforming and analysis logic (figure 4.2). Probes are connected to hardware signals, normally on the back plane, so that the activity of these signals can be monitored. These signals are fed to the transformation logic, where the subset required for analysis and display are filtered out.

Hardware tools are classified according to their flexibility, due to the method of implementation, and the power of their transformation logic. **Fixed** hardware tools are completely hard wired. They are designed to measure specific parameters and are often incorporated in the initial design of the machine. In the latter case they are called **internal tools**, in comparison to external tools which are added and removed as needed. Timing meters (figure 4.3) and counting meters (Svobodova 1976a) are typical fixed hardware tools. A **timing meter** measures the duration of an activity, for example - channel busy, by sampling the state of a signal associated with that activity. Thus, timing meters can be used to measure execution time and utilization. **Counting meters** count the occurrences of events, for example - count all the references to a memory bank. Counting meters can be used to measure execution frequencies and throughput. Sometimes, the event to be counted is generated by an **event trap**, a device which detects the occurrence of an event and generates a pulse to the counter, for example - detection of

subroutine calls. The information accumulated by a meter can usually be read by an operator from a display, and in some cases, it can be read by a program. The simplest fixed hardware tool is the CPU wait light.

Wired-Program hardware tools include a logic plug-board that allows a variety of boolean and counting functions to be implemented in the transformer section. Event filtering can be changed by rewiring the plug-board. In some advanced systems (Murphy 1969) associative memory is also used to detect events. Wired-program tools are normally **external tools**: free standing devices that sense electronic signals in the circuitry of the measured system, and record them externally to the measured system.

Counters can be used to implement timing meters and counting meters. Logic gates can be used to combine and sequence signals. Comparators and sequence detectors can be used to trigger recording devices, to produce event traces. The results of the event filtering can be displayed, or saved in a data base for later analysis.

In **Stored-program hardware tools** the logic plug-board is replaced by a computer-controlled event-filter. Filtering functions are set up by software, not by manual insertion and removal of modules in a plug-board. The logic-state analyser, discussed in chapter 5, is the latest development in stored-program tools (McKerrow 1983). The ability to control filtering with software gives stored-program tools greater flexibility, and if an interactive user interface is provided, greater ease of use than wired tools. Some early tools, however, did not have the resolution of wired tools.

4.3.1. Characteristics of Hardware Tools

Many things that are of interest to performance evaluators can be measured with both hardware and software techniques. Hardware tools have some advantages relative to software tools, and some disadvantages:

- Hardware tools cause little, or no, interference to the system being measured, and thus, they can be used for long periods.
- They have high resolution - often greater than the clock frequency of the system under study.
- Simultaneous measurement of overlapped activities is possible.
- Some hardware related activities are accessible to hardware tools but not to software tools - for example data transfer in a buffered peripheral.
- Hardware monitors can be used on any system, provided the relevant signals are accessible.
- Software related events can only be sensed when they are accompanied by an instruction at a known address.
- The state of a memory location can only be monitored during read or write operations to that location.
- Due to their high resolution and low interference, hardware tools provide very precise readings.
- Attaching hardware probes to a computer makes maintenance men nervous. Poorly installed probes can load signals, hence impacting upon performance.
- Probe attachment takes time and skill. Improperly attached probes produce misleading information. A means of verifying that the readings are correct is essential.
- Hardware tools can be used to monitor the failure and restarting of the target system.

4.3.2. Some Actual Hardware Tools

The first hardware monitor was built by IBM in 1961 (Warner 1974) to measure the amount of system time spent in the I/O channels, and how much time the CPU spent waiting for I/O operations to finish. The monitor included a set of six digital counters and some wired logic.

It was physically large, and required an interface built into the 7090 computer. The counters could be used to count events, such as instructions executed, or to measure the elapsed time of functions, such as channel usage. The totals were accumulated in electromechanical counters. Counters were reset by hand at the start of an experiment, and the information in the counters at the end of an experiment was copied down by hand. Within a year, a streamlined version, known as the 7090 **channel analyser**, went into production. About two dozen were built.

In 1962 a different tool, the **program execution monitor** (Apple 1965), designed to do a full program trace to find where and when loops occurred, was built. Data recorded in buffers was transferred to magnetic tape. However, even with data encoding, the data transfer rate to magnetic tape was slow compared to processor speed, and once the buffers were full data was lost. A second problem was the time taken to reduce the data: a 5.5 minute recording took from 2 to 7.5 hours to reduce. Data reduction programs produced execution traces, module execution times (where a module was defined by specifying a start and end address), module execution ~~frequencys~~ frequencies, and a graph of the range of instruction addresses during a time period verses time. A more sophisticated version, the **program oriented evaluation monitor** (POEM) reduced the quantity of data by event filtering, but it required too much knowledge of the system for general use.

The **program event counter** (PEC), built in 1964, was an enhanced version of the channel analyser and included an important new feature: a removable logic plug-board that could be programmed by wiring. Signals coming into the plug-board could be routed through AND/OR logic to perform various functions, the results of which were summarized in counters. Events could be timed, or counted, simply by appropriate wiring of the plug-board. In addition to channel operations, channel overlap with other channels and the processor could be measured.

Another feature introduced in the PEC was the ability to test for an address equal to a specified address or within an address range. This was achieved with three pairs of address comparators, which were used to compare preset values to the contents of the memory address and data registers. With this tool, module execution time and module execution frequency (note

this was for code at known memory locations) could be found. Data reduction was still a problem.

The **systems analysis measuring instrument** (SAMI), built in 1967, was a bigger and better PEC. It had 50% more electronics than the IBM 360/50 and weighed 4 tons. With it, the idea of the sensor or probe, which could measure virtually any available signal on the system, not just those provided by the hardware interface, was introduced. A smaller version of SAMI, the **basic counting unit** built in 1968, used a probe as the only interface to the system. In 1969 the **system utilization monitor** (SUM); a portable version (TV Size) of SAMI with 20 probes, plugboard, magnetic tape, and analysis programs; was released as a product to the general public by Computer Synectics of California. Arndt and Oliver (1971) discuss the use of the SUM monitor in the evaluation of a real-time system used in satellite command and control.

Another interesting experimental device, developed at IBM in 1965, was the **execution plotter**. A cathode ray tube (CRT) was used to display a real-time graph of program execution: memory address verses elapsed time. It provided a good overview of program execution but it was limited by the CRT technology of the time.

Another monitor, the **Time Sharing System Performance Activity Recorder** (TS/SPAR), was built to study the IBM System 360/67 time sharing system (Schulman 1967). It was able to monitor up to 256 simultaneous signals, reduce them to a maximum of 48 measurable events, record periodic summaries of these events on tape, count events and measure their duration, interrupt the CPU when a specified state occurred, and respond to CPU-generated control signals which automatically enable/disable the measurement process or serve as flags. The ability of the target processor to control the measurement tool was a new idea. Data reduction facilities included counters for accumulating event parameters and comparators for dynamic monitoring of data paths. Schulman (1967) discussed a number of the significant computing developments where measurement was needed (multi-tasking, multi-processing, virtual memory allocation, dynamic address translation, re-entrant code, I/O handling), but whether measurements were ever made in these areas with TS/SPAR has not been reported.

The **system logic and usage recorder** (Murphy 1969) used associative memory to reduce the amount of data available to the set desired for analysis. The associative memory could be instructed to record data only if it was new. The basic associative processes of interrogation and storage were extended, by means of a system of data routing and field control, into a capability for performing advanced data reduction with data processing algorithms. The algorithms reside in a control storage. For example, when measuring the time a program spends in various areas of memory, the program counter is fed to the associative memory, which compares the program counter to the address ranges already recorded in the associative memory. If the address is in a range that has already been seen by the memory, the appropriate counter is incremented; if not, a new counter is automatically assigned to the new memory area and incremented. At the end of the experiment, a set of module (memory block) execution times is read from the memory. Murphy (1969) discusses a number of applications of this very powerful tool.

Researchers outside of IBM have also been active in the development of hardware tools. Roek and Emerson (1969) built a hardware monitor for the specific purpose of **monitoring program flow** by recording and profiling the execution of jump instructions. Restricting the measurements to jump instructions significantly reduces the amount of data. Further reduction is done on the fly by detecting simple loops and recording the location of the first loop jump, the number of loop jumps, and the duration of one pass through the loop. These techniques illustrate the wisdom of understanding the operation of the object you are trying to measure. Further data reduction, done by software, produces graphical (plot of address versus time) and statistical information (utilization of modules).

The **Neurotron monitor system** (Aschenbrenner et al 1971) is considered by many to be the penultimate hardware monitor. This monitor combines the best features found in previous monitors; overcomes the stated deficiencies in many previous hardware monitors; and provides interaction by operator, or program, with the data accumulation, analysis, and display processes. The monitor was based around a minicomputer with a multi-tasking, priority,

operating system. Data reduction, and event filtering, logic was configured under program control. A small plug-board was included to aid in I/O selection. Programmable registers control the selection of input signals, logic functions, sequence detectors, counters, signal paths, and control functions for a particular experiment. Counters, or groups of counters, could be enabled and disabled under program control. As a result, sampling intervals are completely at the discretion of the programmer.

Sequencers are logical devices used to detect the occurrence of an event relative to previous or subsequent events. Each sequencer is designed to accept pulses representing an event (addresses, device movements, etc.), to track subsequent events, and to detect breaks in defined sequences of events.

A key element in the acquisition of data is a random access memory (RAM) with its associated arithmetic unit. When gathering statistics the RAM can be used as a set of accumulators (timing and counting meters). Since the RAM can be loaded either from the monitoring processor or from the target system, it can also be used as a set of comparators.

Once data has been retrieved; from counters, RAM, sequencers, or probe registers; information buffers may be updated and recorded, and displays generated. By use of rotating buffers, double buffering, and other techniques, statistics with a resolution a few milliseconds, involving significant filtering and compression of data, can be recorded and displayed. Amiot et al (1972) discuss the use of this monitor to evaluate a remote batch processing system.

The **ADAM hardware monitor** (Shemar and Robertson 1972), built for use with the Xerox Sigma computer, is a specialized mini computer. Use of a solid state associative memory and parallel instruction execution gives ADAM the capacity to monitor the Sigma system at a ~~precision~~ resolution comparable to the instruction execution rate. An ADAM program for an experiment is developed on the Sigma system. Then, the compiled program and preset data values are downline loaded into the ADAM monitor.

The SIGMA system controls the measurement program running on the ADAM monitor,

and periodically the acquired data is transferred from ADAM to SIGMA for analysis. Collins (1976) reports on the use of the ADAM monitor to localise and identify performance problems in an overlay-loader.

Fryer (1973) describes a **memory-bus monitor** which was designed to assist in program development on dedicated real-time computers. The memory-bus monitor was a piece of hardware attached to the bus connecting the central-processing-unit and the memory. In many ways it is the forerunner of the modern logic-state analyser.

A simple memory-bus monitor could be used to display the contents of a specified memory location as it was read or written. A more sophisticated version included an address stack. Each address that appears on the bus is pushed onto the stack. When a selected address appears on the bus the stack is frozen and its contents displayed, effectively providing a 'come from' trace.

The address stack was replaced with a content-addressable memory that could be used either as a stack or a set of comparators. A data stack and a timer were added, and the number of address comparators increased. The resultant monitor could be used to:

- look at bus traffic when the computer was operating in single step mode,
- detect and stack accesses to specific memory locations,
- stack bus activity, and halt stacking when an activity of interest occured (the stack contained a trace of bus activity immediately prior to the activity of interest),
- time blocks of code, e.g. loops, and
- count the number of times a branch was taken, an opcode was executed, or a memory location referenced.

Deese (1974) sought a solution to some of the **operational problems** with hardware monitors. These include: the difficulty of locating test points, signals not available at test points, signals not synchronised, the adverse consequences of poorly attached probes, the problems involved in

changing probes, and the increasingly large number of probes required. To overcome these problems, when monitoring the fire control system in the Trident submarine, he introduced a new concept: the **monitor register**. All the signals required for monitoring were wired to eight input registers. Under microprogram control, at every minor cycle, the contents of one of these input registers, selected by a bit pattern in the microcode, is transferred to the output register, where the external hardware monitor can read the information plus the number of the currently selected input register. Thus, by designing performance monitoring features into the machine, the operational difficulties of using a hardware monitor have been significantly reduced.

Hempty (1977) discusses the use of a hardware monitor, a **channel utilization monitor**, to evaluate the performance of the IBM 3850 mass storage system. The monitor, which attaches to the channel interface, collects data about each interaction between the channel and the devices attached to it. An event trace of channel interactions (channel command or device response to a command) is produced. Each interaction record includes: the number of bytes transferred, the time taken by the channel to execute the command, a command identifier, channel status bits, the device addressed, and a time stamp. Evaluating this data to detect bottlenecks was an enormous task, so the trace was expanded to indicate the type of activity the device was carrying out (for example seek or read).

4.4. Software Tools and Techniques

Software tools consist of instructions added to the target process to gather data related to its performance. Generally, no additional hardware is required. The instructions added to the code (called software sensors or software probes) collect the data, reduce the data, store it in internal buffers, and transfer the buffer contents to backup store. Analysis and display of results are usually carried out at a later stage.

The most common use of software tools is in the generation of system accounting logs. These logs are used by managers for capacity planning, and in the creation of customer accounts. The simplest log consists of a sequence of messages that indicate the start and termi-

nation of activities, the time of day, and the job name to which the activity is related. More sophisticated logs include the amount of CPU time, the amount of I/O time per device, and the memory usage of individual jobs. An accounting program is then used to collect job related information for each job, and to produce customer accounts.

Analysis programs can produce a complete record of the day's activity, and calculate system statistics (for example the utilization of various resources and the throughput for different job classes), from the logged data. Careful use of system logs can provide a lot of the measures needed for evaluation. However, these logs are normally designed for accounting purposes, often with extra information thrown in for interest, and thus, may not include some desirable measures. Another log, provided by some systems, is a report to the user, at termination of his job, on the real time and cpu time used by the job.

A useful debugging tool, found in some software tool kits, is a program-execution profiler. Prior to compilation, a preprocessor inserts checkpoints to detect variable accesses and flow-of-control branching. During execution, these checkpoints increment counters. After execution a profile, showing either the number of times each line of code was executed or the number of variable accesses, is displayed to the user.

All software tools are event driven. **Sampling** tools execute in response to external events. When an event occurs, the state (often the operating-system tables) of the target process (system) is read and recorded. Collected data is analysed using statistical methods. Sampling techniques reduce the amount of data needed to estimate some quantities. In a sense, they are equivalent to taking the pulse of a system.

Sampling can be done at periodic or random intervals. In time sampling, the event which initiates the sampling process is the termination of a specified time interval. In count sampling, counting meters initiate the sampling process, when a specified count is reached, for example every n disc accesses. The counting meter can be a hardware device, such as those discussed in section 4.3, or a software device: a memory location that is incremented every time a procedure

executes.

Event detection is done by the sampling program periodically checking the value in the meter. When the meter reaches the desired value, the sampling program resets the meter and records the desired state information. Events can also be detected by the meter routine, which then calls the sampling program. In this case we have an internally-driven tool. The essence of sampling is that the measures are synchronised to the termination of a sampling period, not to changes in the internal state of the system.

Internally-driven tools, by contrast, execute in direct response to the occurrence of events within the system. Thus, they are synchronised to the internal state of the system. Internal events are all detected by the execution of a piece of software, called a checkpoint or software probe, inserted into the target process. In an interrupt-driven operating system, each invocation of a module of the operating system is caused by an interrupt or a trap (supervisor call or software interrupt). These interruptions occur in response to changes of system state, and cause changes in the system state. Thus, a logical place to put checkpoints is in these routines. This is a neat way of decomposing the system object. Monitors based upon checkpoints in interrupt routines are called **interrupt-intercept monitors.**

Checkpoints can be inserted at various points of interest in the target process, not only to detect the execution of an instruction, but also, to detect a data structure being updated or a variable being assigned a certain value. Thus, they can be used to get at software specific information; for example user name, disc-file name, variable contents, job class, etc. In order to use system resources, read system tables, or get around system security, a checkpoint may have to use a supervisor call. A checkpoint can either collect data itself, or call a measurement program for the same purpose. Having recorded the required data, the checkpoint returns to the calling process. The data is usually stored in a set of in memory registers. When these registers are full, or when the experiment is complete they are transferred to backup store.

A **fixed software-tool** is a permanent checkpoint which collects data every time that sec-

tion of the target process is executed. Accounting routines use fixed tools. Software meters are usually fixed tools. A software timing-meter increments a memory location when the clock routine executes if a flag is set, for example user time is incremented while ever a user process is executing. One way of minimising the overhead associated with incrementing software meters is to implement the counting routines in firmware (Blake 1980).

Tools which can be enabled and disabled, or inserted and removed, at will are **non-fixed** tools. In some non-fixed tools the event to be detected can be modified, or the data variable to be recorded can be changed. These facilities enable the user to tailor the measurements to the experiment at hand, and eliminate the execution of unwanted checkpoints, reducing interference. Systems instrumented in this way either have a set of permanently installed tools which can be enabled by a monitoring process or have facilities for automatic insertion and removal of checkpoints.

Some systems (Deutch & Grant 1971) allow the insertion of user supplied checkpoint routines. These must be verified before installation to ensure that they do not introduce logical errors into the target process. To facilitate automatic verification, routines must conform to a standard structure and loops must have fixed indexes. The criteria which must be checked for and eliminated include branches into the target program, use of illegal or privileged instructions, indeterminate execution time, excessive execution time, storage of data outside the defined data collection area, self modifying code, re-entry problems, and violation of security and/or privacy.

Other systems provide a macro facility (Ferrari and Liu 1975) that enables the combination of standard checkpoints into measurement macros, for example a macro to trace the flow of control in a program. Standard checkpoints supported by these measurement systems include: counting meters, timing meters, enable/disable checkpoints, set/clear flags, record time, record event trace tuples, transfer statistics to backup store, and clear monitor registers.

4.4.1. Characteristics of Software Tools

Software tools have some advantages, relative to hardware tools, and some disadvantages:

- Software tools interfere with the system (Bourret and Cros 1980). Checkpoints take time to execute and use up memory. If the software tool uses less than 5% of system resources, then the measurement accuracy is generally adequate. One method of reducing interference, where a writable control store is available, is to implement checkpoints as firmware routines callable as assembler instructions.

- Resolution is lower than for hardware tools. They are most suited to recording macroscopic, infrequent events.

- They can record events only in a sequential manner, and they stop the execution of the target process while the data is being recorded.

- Hardware related events can only be detected if they are accompanied by the execution of a program instruction of the updating of a fixed memory location. Also, peripheral devices can only be monitored through their communications with the central processing unit.

- Software monitors can only be used on the system they were designed for.

- Software related information; for example program name, variable contents, and dynamic data structures; can easily be sensed by software probes.

- The state of memory locations can be monitored at any time by software tools.

- Software tools can only provide rough timing measurements, depending upon the precision of the system clock.

- Insertion of faulty software probes into a system can cause program faults, and may even cause the system to crash.

- Software tools are usually easier to install than hardware tools, particularly for programmers, and may be more flexible. Also they generally cost less.

- Changes to an operating system can drastically affect the accuracy of software tools, requiring compensating modification to the software tools.
- Software tools can handle dynamic environments which create problems for hardware tools: relocation of code modules, virtual memory, recursion dynamic data structures, and interpretation of programs.
- To use a software tool, the system must be instrumental to the level that pauses in the execution of the target process, due to an external interrupt, etc., are detected. A hardware tool can pick this, simply by detecting references to address outside the program area.
- A major headache with software tools is verifying their accuracy. Carlson (1977) discusses the use of the SUM hardware monitor to verify the CUE software monitor.

4.4.2. Some Actual Software Tools

Who did what first is not as clear with software tools, because many people wrote simple tools to help with their work and never reported them. As a result, they became part of the folklore of computing. Also, some notable instrumentation systems (for example, those in Multics and Cm*) are not included here, as they are covered in later chapters.

Herbst et al (1955) used software tools to measure the **instruction mix** of Maniac programs. Estrin et al (1967) proposed the use of **checkpoints**, which increment counting meters, to produce program execution profiles. These checkpoints were to be added by a preprocessor, prior to compilation of the Fortran programs.

Knuth (1971) popularised the ideas of **program profiles** and **instruction mix measures** for high-level languages. He used two approaches to dynamic program analysis. In the first approach, a program profile was produced by inserting **meters** into the program, at appropriate places. Using this profile, in conjunction with a knowledge of the instructions used in the program, an instruction-mix table was produced. His second approach involved **sampling the program counter** at regular intervals, and counting the number of times addresses within specified

address ranges (32 byte) were detected. At the end of a program run, a histogram (number of samples per address range) was plotted. By comparing the recorded addresses to the Fortran-program address-map it was possible to get a coarse utilization profile.

Snuper Computer (Estrin et al 1967) was a system proposed at UCLA. How much of the proposal was realised is not reported. The proposal included the production of a number of graphical displays, similar to those in chapter 2, for data reduction and analysis. Three development phases were proposed.

In phase 0, the self measurement phase, source programs would be instrumented with a procedure which built an execution profile. Bussell and Koster (1970) discuss two self measurement tools that were implemented as part of the UCLA computer instrumentation project: a self-simulator that closely duplicates the operation of the machine it is running on, and a program for making precise measurements (2 microsecond accuracy) of the time duration of events.

In the first tool, a data gathering routine collects instruction mix statistics, while the target program is being executed on the simulator. Using a simulator enables complete instrumentation of the target process by instrumenting the simulator. In the second tool, a time subroutine, callable from a user process, starts and reads a precision hardware clock.

In phase 1, the output of the instrument procedure was no longer written into an array, but was written to an external tool via an output port (a hybrid tool). In phase 2, sufficient data about the program is produced by the compiler and sent to the external tool, to enable a purely hardware monitor to examine the operation.

The idea of **passing information about the program to the external tool prior to execution** is one contribution made by this project. This idea was taken up and extended by Plattner and Nievergelt (1981) during the development of a program-execution monitor. Estrin et al (1967) were the first to propose the concept of a **hybrid monitor**, a name invented later, where both hardware and software tools combine to measure a system. Thus, they contributed some valuable ideas to the field.

Cantrell and Ellison (1968) measured the performance of the **General Electric GECOS II multiprogramming operating system**. The operating system consisted of 64 programs. The GECOS system keeps a running total by program (system and user) of all processor, channel, and device, times for accounting purposes. These totals are updated for each period of processor use, and for each I/O transaction. When a user program terminates, all of its accounting times are transmitted to an accounting file, and the timing counters reset.

Thus, the system is very highly, and very accurately, instrumented. In trace mode, a trace entry is made in a circular list for every major operating system event, such as handling an interrupt or dispatching a process. Analysis of the user program accounting data did not give a clear picture of where the time went. Analysis of dumps of the counters for the operating system programs (the operating system data was not included in the accounting report) indicated that overhead processing was small, and idle time was high. Trace mode was turned on, and trace entires recorded on magnetic tape. Unfortunately, during five minutes execution 350,000 trace entries were recorded, producing a significant data reduction problem. A succession of data reduction and analysis techniques were tried, and two methods were finally settled upon.

A program, called MAPPER, samples most of the timing counters every two seconds, calculates the changes since the last measurement, calculates the percentage of processor and channel time used by each process (system and user), and prints these percentages. In addition, run averages are printed, and, at program initiation or termination, the name and core map of the current program are printed. In an alternate mode of operation, MAPPER records the complete event trace, which is processed to produce an event report. Another option enables the measurement of the distribution of compute time within a program.

Here we have a combination of software techniques: timing meters, event detectors, event trace, and sampling; all in the one system. In addition to looking at the interaction between programs within the operating system, the tools were also used to study the execution of individual programs. The results of this study were: fourteen performance bugs were found and fixed, an average throughput improvement of 30% was achieved, and their understanding of

how the operating system really worked was increased.

Campbell and Heffner (1968) discuss additional techniques used when measuring a later version of the operating system, **GECOS III**. They found that when designing a system for measurement, it is important to be able to measure the length, and wait time, of system queues, for example the dispatcher queue.

One problem they had was inferring what had happened prior to a system failure. When they studied the system, they found that all communication between modules passed through a common routine. The routine was modified to record information about each intermodule transfer, in a circular list. Thus, at any time, a trace of the latest transfers could be obtained, and from this the operation of the system could be summarized.

This trace facility was a tremendous advance in easing the job of analysing system failures, but the interference was high and often important information was lacking. An event trace, which records the occurrence of system events (over fifty), was implemented to overcome these shortcomings. Events are traced by routines inserted into the systems code, which record data into a circular list. Traces for individual events can be turned on, or off, at system start-up time.

An idea introduced to help find system bugs was **system auditing**. To "audit" means to check and verify. Audit programs were written to check operations on system queues, to check-sum critical tables every time they were referenced, and to checksum all system files as they were loaded into the system. These audits quickly pinpointed the occurrence, and cause, of data corruption in these system critical areas.

The **System Internal Performance Evaluation program** (SIPE) recorded data about significant events that occurred in the IBM System/360 Time Sharing System (Deniston 1969). Events are detected by **hooks** inserted at strategic locations in the supervisor. When flow of control reaches a hook, an interrupt is generated transferring flow of control to the SIPE monitor which collects the desired data and then returns flow of control to the supervisor process. A

simple mechanism exists for enabling and disabling hooks, such that they are either all on or all off.

Balzer (1969) developed the **EXDAMS system** as an extendable debugging-tool for high-level-language programs. First, the source program was analysed to build a model of the program, and to insert probes. During execution, these probes produced an execution history of the program. In general, the history contained all the dynamic information needed to update execution time either forward or backward, while the model contained all necessary static information. A number of static, and motion-picture, displays were available for studying program execution and for debugging programs. The motion-picture displays allowed the operator to watch the execution of the program, in slow motion, either forwards or backwards.

The **Informer** (Deutch and Grant 1971) was meant to attack three problems that arise continually in large programming systems: debugging, performance analysis, and environment analysis. Users were allowed to name an arbitrary point in the operating system (a checkpoint), and to specify a measurement program which was to be executed, in the environment of the operation system, each time the flow of control reached that checkpoint.

To ensure that no bugs were introduced, user measurement programs were tested to see that they did not modify the environment, and that they executed within a specified time. Tools were provided to minimise the time and effort involved in composing, debugging, and executing, measurement programs. Attaching the measurement routine to a checkpoint was not as easy as it may appear. The user had to have a way of specifying the checkpoint location, the loader had to have a map of spare memory into which it could load the measurement routine, the integrity of register contents had to be maintained, and the target process had to be patched.

Two commercial software monitors were developed by Boole and Babbage Inc (Holtwick 1971). One, called the **configuration utilization evaluator** (CUE), was oriented towards the systems programmer; the other, called the **problem program evaluator** (PPE), was designed principally with the applications programmer in mind. Kolence (1972) developed his Software Physics

during the development of these tools. Both tools consisted of two parts: an extractor which sampled, via a random sampling-technique, for user specified data, and an analyser which transformed the extracted data into reports.

The PPE extractor ran as a normal OS/360 job, and collected performance data on a program as it executed. The program under study was sampled at a rate specified by the user in multiples of 1/60 of a second. At each sample, the absolute program address, the program status (waiting, executing, etc.), the program name, and the overlay segment number were recorded. PPE reports could be used to pinpoint high-usage areas in the program, and areas using excessive time.

The CUE monitor ran as the highest-priority task in the system, and measured the system as it operated. CUE also sampled in multiples of 1/60 of a second. It collected detailed data concerning equipment utilization, queue lengths, mechanical-access movements on direct access storage devices, supervisor transient-load-module loads, distributions of I/O requests, and distributions of transient areas in use. CUE could be used to investigate the cause of bottlenecks, job scheduling strategies, multiprogramming contention, the effect of file placement upon access time, etc.

The response time of the **OASIS system** was claimed to be slow. OASIS was a system developed at Stanford to support both batch and interactive modes of operation on an IBM 370. A software monitor (**OEM**) (Svobodova 1973a) was built to measure the scheduler and the OASIS resource (tables of blocks which are allocatable to online tasks) routines. The executive was instrumented to maintain counts of the resources in use, for each of the four OASIS resource types. Also, the number requests for an OASIS resource which could not be granted, because the resource pool was empty, were counted. These counters were sampled every sixty seconds and, the current values recorded in a buffer. After twenty records were accumulated, they were transferred to disc together with an array containing scheduler information. At the end of each scheduler pass, a counter SQ_{ij} was incremented, where i is the scheduler-queue length, and j is the number of tasks on the queue the scheduler interrogated before it found one

which could be dispatched.

Measurements were made over a period of weeks, the arrays were combined to produce scheduler statistics for the period, and the counter information used to detect exhaustion of resources. The measurements confirmed that the resource pool was too small, hence causing deadlocks when task requests could not be completed due to lack of resources. Studies of the system when no on-line tasks were executing enabled the measurement of overhead (11 to 12%). A complementary hybrid tool will be discussed in the next section.

The **Mesa Spy** (McDaniel 1982) is an interactive, sampling tool that gathers real-time performance-data for other, independently written Mesa programs by using an extension of the **program-counter-sampling** technique. Ingalls (1972) and Rafii (1981) have also used the program-counter-sampling technique reported by Knuth (1971). The Spy is an independently-compiled Mesa-module which is loaded into the system with the target program. The Spy does not contain built-in information about the target program, but exploits knowledge of the Mesa language and run-time environment. Information in the run-time call-stack is used to determine what code is responsible for the resources being consumed.

The Spy provides an interactive, symbolic user-interface; the user describes the program by module, or procedure, name; and the Spy provides symbolic output at the module, or source, level. Optimising compilers modify the mapping between the program counter and the source language, hence, interfearing with the Spy's ability to print analysed data in terms of source-level statements.

The Spy has two modes of operation: ~~course~~ coarse-grain-data collection, and fine-grain-data collection. In the ~~course~~ coarse mode, the Spy collects information about program execution on a module basis. Every module (compilation unit) in the system has a module number (an index into a special table that belongs to the Mesa run-time-system). The Spy maintains an array of data-collection buckets, indexed by module number, as its basic data structure. Each bucket contains a counter and a pointer. The counter is incremented if the associated module is running when

the Spy samples the state of the system.

A **charge-the-caller** flag causes the Spy to follow module nesting back up the call path until a bucket is found where the flag isn't set. The counter in this bucket is incremented, and thus, the resources used by the currently executing module are charged to the calling module. With this facility, execution of system-library modules etc. can be charged to the module which uses them, giving a truer picture of resource utilisation. By switching the flag on and off for successive runs of the same experiment, and by switching fine-grain sampling on and off, excessively used routines can be pinpointed and analysed.

The pointer in the data-collection bucket points to a data structure used for fine-grain collection, or is nil if fine-grain collection is turned off. The fine-grain data-structure contains a counter, and a charge-the-caller flag, for each non-overlapping program-counter interval in the module's code-space.

Systems that have been designed for measurement often have a number of complementary tools. The PRIME multi-processor system (Ferrari 1973) includes a general purpose software measurement tool (SMT) (Ferrari and Liu 1975). This tool is discussed in section 9.4.1.

4.5. Hybrid Tools and Techniques

Hybrid monitors (figure 1.5) attempt to take advantage of the complementary nature of hardware and software tools. In a typical hybrid monitor, software tools detect events within the system and write information relative to those events to a hardware interface. Data arriving at the hardware interface is recorded, together with other hardware signals, by an external hardware tool, where it is analysed and displayed. Not all hybrid tools fit this pattern, a wide range of variations is possible. Hybrid monitors have the potential of being able to measure all the information about a system, with the precision of a hardware tool, but with less interference than a purely software tool. In a sense, a hybrid monitor is an intelligent peripheral, or at minimum an intelligent alternative to a backup store.

The major design decision to be made when developing a hybrid monitor is: which section should measure what? If you do not have a well thought out philosophy of hybridisation, the result may turn out to be an ad hoc collection of dissimilar components rather than an integrated measurement system. This question is analysed in the context of the performance measurement formulation in section 6.4.1. Answering this question involves selecting a combination of hardware and software tools, determining the power of each tool, dividing data reduction functions between the tools, specifying the communications protocols between the tools, and deciding what data analysis and display techniques are to be used.

Some hybrid tools are simply a combination of components from existing hardware tools and software tools; others are designed with a specific hybridization goal in mind. Hybrid tools can be loosely classified according to the partitioning of functions between hardware and software. All tools aim to keep the interference of the software tool to a minimum. Some however, do this only during the execution of an instrumented process and use considerable target system resources to set up the target process.

These tools use a modified compiler to produce an extensive model of the target process, which, together with a load map, is passed to the monitoring computer before the target process is executed. During execution, the monitoring computer (note the assumption that the hardware section of a hybrid tool includes a computer; this is generally true but not always) traces the execution path of the target process, and records variable assignments, using purely hardware techniques. The hardware measurements are combined with the process model to produce a detailed execution history. In this style of hybrid tool, the two computers act as communicating processors and measurement is usually controlled by the monitoring tool. For the monitoring tool to have total control over the measurement experiment it must be able to request the execution of processes on the target processor.

A second approach, which also produces zero interference during the execution of the target process, is for the loader to pass a load map of the target process to the hardware tool. During execution the hardware tool collects the specified data. When execution terminates, the

recorded data is passed back to the target processor for analysis. In this case, the hardware tool is an intelligent data reduction and capture device under control of the target processor - an intelligent peripheral. This tool is usually more restricted in the range of measurements that it can make than the previous tool.

If some interference is acceptable, then other approaches are possible. A hybrid tool can often measure the interference and compensate for measurement errors. One approach is to pass the target process through a pre-processor which adds checkpoints before compilation. During execution, the checkpoint routines pass information to the hardware tool. The above methods are all suitable for dynamic analysis of program execution but have short-comings when instrumenting a complete system, especially in dynamic-memory-allocation environments.

The approach most suitable to complete systems is to add fixed checkpoints at design time. This approach produces some interference during execution, but in the case of system processes, it eliminates the use of system resources to set up the instrumentation, except at system generation time. Software probes are used to detect events, and to pass event descriptors and stimulus information to a hardware tool, which does the rest. It is often cheaper to leave fixed checkpoints in all the time than to test enable/disable flags. Thus, a continuous stream of information is presented to the transformer section of the hardware tool. The monitoring computer has complete control over measurement experiments, however, at system generation time a checkpoint table must be produced. Fully instrumented systems often include a combination of tools to enable measurements at all levels in the object hierarchy. Determination of what constitutes a checkpointable event is based upon the designer's formulation of measurement and the purpose for which measured data is to be used.

A major consideration in the use of hybrid tools is the selection of sensors. Some of the criteria to be considered when selecting the sensors to be used are:

- Your formulation of measurement and your philosophy of hybridisation.

- The number and type of quantities to be measured.
- The sensors which can be used to measure each quantity.
- Of the sensors available, which one has the most advantages and the least disadvantages.
- Is data reduction to be done at the sensor or in the external tool.
- Can the sensors be installed easily.

Hybrid tools add a new element to our basic measurement tool configuration: the interface between the software and the hardware tool. This interface can be as simple as a parallel output port, or as complex as a direct memory access channel. Some interfaces are passive, they present information to the hardware tool under control of the software tool; others are dynamic, the hardware tool can request information from the software tool. Some interfaces allow the hardware tool to pass information to the software tool, or to interrupt the target process. These features are useful for feedback control of operating systems, and in debugging environments, where the analyst may want to stop and restart the target process.

Data transfer from the target system to the monitoring system can occur in response to an interrupt generated by the interface when it is written to by the software tool. Alternatively, the hardware tool may sample the interface at regular intervals. Data reduction can be done by looking for specific bit patterns, or pattern sequences, in the data. The hardware section of a hybrid tool usually includes a computer. Data analysis, display, and as much reduction as required is done by programs running on this computer.

4.5.1. Some Actual Hybrid Tools

The concept of a hybrid tool seems to have originated with the **Snuper Computer** proposal (Estrin et al 1967). In phase 1 of Snuper, hybridization was proposed as a means of reducing the interference of the software tool used in phase 0. The proposed hybrid monitor would work in the following manner when implemented. Checkpoints (called emitters) were still added to the

target process by a preprocessor. The event data is no longer summed into an array by software, but is written to an interface. The interface calculates an address in the monitoring computer's memory, from the emitted event descriptor, and increments that location. At the end of the experiment, a table of event counts (module execution counts) exists in the monitoring computer's memory rather than in the target computer's memory. During the measurement experiment, the monitoring computer produces dynamic displays of event activity.

Also, flags in the event counters, set by software, could be tested during the incrementation phase, and further recording disabled or interrupts to either target computer, or monitoring computer, generated. Thus, all sensing is done by software, but data reduction is done by a combination of checkpoint placement and hardware techniques. Data recording, analysis, and display are all done by the external tool. Although this instrument was primarily designed for monitoring the execution of programs, it could be extended to other levels of the hierarchy by providing tools to enable event description and checkpoint insertion appropriate to the level.

Phase 2 of the Snuper proposal sought to reduce interference to zero during the execution of the target process. To do this, checkpoints were no longer inserted into the target process. Instead, the compiler and loader were modified to provide sufficient information about the target process for the monitoring system to set up the significant event filters. The data collection and reduction circuits, in the external monitor, had to be modified to handle the new types of data.

When implemented, phase 2 would work in the following manner. All data appearing at the hardware probes (presumably the program counter) is clocked into a first-in-first-out queue. As the data leaves the queue, it is used to address a bit in an event-table memory. If the bit is set, the data relates to an event of interest and is to be recorded; otherwise it is discarded. For each bit set in the event table, there is a pointer to an element in a data accumulator array. When an event is detected, this element is incremented. Several events may increment the same array element, for example execution of instructions within a loop or calls to procedures. A second event filter can be used in conjunction with the first to treat pairs of events as one event.

For example, detection of the start and end address of a module indicates that the module has been executed once.

I have been unable to ascertain how much of this proposal has been implemented. Bassell and Koster (1970) discuss two self-measurement routines, but these are different to the self-measurement procedure in phase 0 of the proposal. The ideas developed in the Snuper proposal have been utilized by other researchers in the development of hybrid monitors. A hybrid tool (Grochow 1969) was used in the instrumentation of **Multics** (section 8.1).

The **SPY monitor** (Sedgewick et al 1970) was developed at Western Electric to evaluate the operation of OS/360 MVT executing on an IBM system 360 model 50. A parallel data path connected the target processor to the monitoring processor, a PDP 9. In addition, one hardware probe monitored the state of the wait light on the target computer.

A software tool, running on the target processor, sampled selected information, accumulated statistics, and transferred the collected data to the monitoring processor at selected intervals. The external tool monitored the state of the wait light, formated all information for display, and produced a hard copy record of important statistics.

By replacing the backup store normally associated with a software monitor with a monitoring computer, real time analysis and display of the dynamic action of the target system became possible. The CRT display was split into four sections: one containing specific information on the jobs being processed, one showing the utilization of the direct access devices, one containing running averages of selected summary information, and one containing a continually recycling graph of CPU utilization.

The software tool was a sampling tool with three defined sampling intervals: one for sampling internal information, one for gathering statistics, and one for data transmission. Information about individual tasks was obtained from the tables in the task queue. Information about direct access devices was also obtained from operating system tables. Thus, the data was collected from tables within the system by sampling programs running under the operating system,

without having to modify the operating system. This approach requires an intimate knowledge of the operating system, and can only produce coarse measures.

Nemeth and Rovner (1971) took a completely different approach in the instrumentation of the **TX-2 computer** at MIT Lincoln Laboratory. The TX-2 was a time shared system with extensive facilities for supporting interactive graphics. Each user of the system had a virtual computer, which appeared to have all the facilities of the actual system, to herself.

Nemeth and Rovner's aim was to detect conditions in the user's virtual computer, and to interrupt the virtual computer when an event of interest occurred. Events of interest included: the execution of n instructions, supervisor calls, subroutine calls, and references to particular variables. In response to the interrupt, a routine was called to collect and process information about the state of the virtual computer. When the interrupt routine terminated, the virtual process resumed.

The TX-2 was equipped with extensive hardware-maintenance facilities. About 160 of the internal signals were wired, in functional groups, to a maintenance probe. The values represented by these functional groups could be compared to values stored in switch registers. The outputs of the comparators (7 of these) could be ANDed with up to 39 other signals to detect an event of interest. The output of the AND gates (two 40 input gates were available) could be used as the sync input to an oscilloscope. They were also fed to event counters. Pulses from the AND gates, and pulses generated upon event counter overflow (i.e. n events counted), interrupted the processor.

Thus, the hardware device could be used to detect events of interest, and to call an internal measurement-routine to record the state of the target process. A user of this system required a memory map of the target process, and had to initialise the comparators manually. Arming of the interrupts, and initialisation of the event counters, could be done under software control. Again, measurements could be made of a whole host of system parameters without having to modify the code of the target process. However, a detailed memory map was needed,

which becomes a problem with dynamic memory-mapping, and allocation, schemes. Also, the addition of a hardware tool of this nature to an existing system can prove to be a difficult exercise.

The **CPM-X monitor** (Ruud 1972) was a commercially available tool which expanded the power of standard hardware techniques by gaining access to software related information through a channel interface. The memory of the monitoring processor had three ports: one connected to the monitoring processor, the second connected via a DMA channel to the target processor, and the third connected to the output of a computer-controlled, hardware-event-filter module. The computer used in CPM-X was a Microdata 1600-D dual processor minicomputer with three-ported memory. Some of the event filtering was implemented in microcode stored in a writable control store.

The target processor could write information directly into the monitoring processor's memory over the DMA channel. To transfer information in the other direction, the monitor interrupts the target processor, which then reads the monitoring processors memory to determine the cause of the interrupt. This facility could be used to request the transfer of operating system tables or event counter contents from the target system to the monitoring system.

The **Burroughs B1700 Computer** (Wilner 1972) includes microinstructions which allow the programmer to write bit patterns to test points accessible to hardware probes. These can be used for indicating the occurrence of events or for passing data to an external tool. The system includes programs to measure and plot module execution frequencies and execution time profiles.

In the previous section the monitoring of the OASIS system with a software tool was discussed. A hybrid tool, the **OASIS task monitor, (OTM)** was also used (Svobodova 1973). The software tool worked well at system level, but it was feared that if it was applied at the task level the interference would be too high.

Each task (terminal) was given a status byte in which the tasks current state was stored.

An interactive task can have a number of states: not logged on, executing, I/O busy, waiting on terminal input, etc. The executive was modified to write the tasks state, and presumably task identifier, to a memory location whenever the state of a task changed i.e. every time the scheduler executed. The machine included a hardware debugging aid: an address-match circuit. The address of the monitoring memory-location was put into the address comparator, in the address-match circuit, with a set of rotary switches. Every time this memory location was written to, a sync pulse was generated, by the address-match circuit, and the external hardware monitor (Computer Synectics SUM monitor) recorded the data on the memory-data bus.

As the state bits were in the same location in the state word for each task, changes in the bits could be counted to obtain event counts of various system activities, for example I/O requests. Also the time between events could be measured, for example the time a task waited for I/O. Thus, by appropriate event mapping into status flags, information about the behaviour of individual tasks and about tasks in general can be obtained. The insight here is, every time the state of any task changes a common piece of code, in this case the scheduler, is executed. Thus, detecting this one event allows the system object to be decomposed neatly into task objects and the task objects into modules. The software probe was ten instructions long; eight were used to update task state, and two to write it to the monitor.

HEMI (Hybrid Events Monitoring Instrument) is an experimental tool developed for use on CYBER computers, during an experiment to ascertain the economic and technical viability of integrating performance measurement and evaluation instrumentation into computer systems (Sebastian 1974). One of the CYBER's peripheral processors was used as the monitoring processor. Peripheral processors have direct access to central memory enabling the peripheral processor to manipulate data in central memory. As peripheral processors are usually used to control I/O channels, the hardware data acquisition front end is attached to a channel.

The hardware data acquisition front end includes probes, timers, counters and sequencers. HEMI reads the contents of the counters, etc. through the I/O channel, and reads the central memory and the central processor registers, and state, by means of the peripheral processor

instruction capabilities. Because HEMI can access both hardware and software data simultaneously, software events can trigger hardware data collection, and vice versa. Data is recorded into a host system storage device. Measurement experiments are set up and controlled from the host processor.

DIAMOND (Hughes 1980) was developed at Digital Equipment Corporation for use in developing software for their family of computers. To gain wide user acceptance, it had to be easy to use and simple to install. DIAMOND includes two processors: a minicomputer for experiment management, including event filter control, and a stored-program micromachine for event filtering and data analysis.

The micromachine was used because a fast (250 nsec resolution), programmable event filter and data recorder was required. In the design of the micromachine, 750 measurement algorithms were classified and analysed. A small set of fundamental algorithms was selected, and an architecture with an instruction set capable of executing the algorithms efficiently was defined. This machine was implemented with off the shelf components.

To be efficient, each algorithm had to be realised in the form of a single-microinstruction loop. That is, in response to each input stimulus from the target system, the micromachine performs all necessary analysis and data-processing in a single instruction step. This mandated a highly parallel architecture.

The program-counter-histogram algorithm captures two distributions in parallel: the intensity of program activity in a given address space is displayed by frequency-count, and by associated execution-time. In terms of the formulation of performance measurement, this is the set of module-execution counts and the set of module utilizations (as times not as percentages), where modules are the individual machine code instructions in the program.

The event-time-histogram algorithm is used to capture a frequency distribution of the time durations between occurrences of two defined events. If the first event is the start of an object and the second is the end of that object, then the time durations identify the paths through the

object, and the frequency distribution is the set of execution frequencies.

Automatic checks are included in some measures to detect values outside the specified range in case the histogram-storage arrays do not have enough entries to store all cases. In these situations, an out of range counter is incremented, and the value which violated the range is saved. This information can be used to modify the experiment specifications, or to detect unusual conditions. Some other measures use the same basic algorithms, for example histograms of data addresses or data values.

The trace-absolute algorithm records successive states of the target system (before, during, or after a defined event or state). State is defined as: task identifier, cpu mode (e.g. supervisor), program counter, and time since start of the trace. A number of other traces are possible: tracing in a circular list until the event of interest (useful for studying crashes), tracing within a task, tracing when the time within a given module exceeds a present limit, and tracing data reads and writes.

A digital interface and switching unit; used to collect, buffer, and multiplex signals from the target processor; is provided for each target processor type (PDP 11, VAX, etc.). This card plugs into the target processor, significantly simplifying probe attachement. The control minicomputer sets up signal paths in the interface to feed the required set of signals to the analysis micro-machine. These signals are masked and compared to range values in four double-comparators. The output of these comparisons is combined to produce a code, which is used as an opcode modifier by the microinstructions executing in the micro-machine.

Software specific information is passed from the target process to the monitor via two registers located in the interface. One register contains the current task identifier; the other can be used by checkpoint routines. Considerable emphasis has been placed on a clean, forgiving, friendly user-interface. Areas of concern in the design of the user interface included: minimal set-up time, natural language interface, semi-automatic replication of experiments, maintenance of a measurement audit trail, meaningful reports, and automatic adjustment to the user's level

of competence.

Ferrari and Minetti (1981b) discuss the design of a low cost monitor, **HPM**, for measuring a minicomputer destined to be a basic component of Olivetti's local computer networks. The HPM is a programmable event filter which transfers recorded data directly into the memory of the controlling minicomputer. The controlling minicomputer programs the event filters, and the dma channel, in response to user defined measurement scenarios.

The address bus, data bus, clock signal, and up to 12 other lines are connected from the host processor to HPM. HPM event filters are very similar in power and function to the event filters found in a logic-state analyser. Software probes write their information to a fixed memory location. The hardware probe can obtain the information by detecting a write to the memory location and then reading the data bus.

Figure 5.1 HP 1610A Logic-State Analyser

5. Using a Logic-State Analyser as a Hardware Tool

As digital logic circuits increased in complexity, traditional fault-finding tools became inadequate. Oscilloscopes can be used to monitor one or two high-speed, repetitive, digital signals, or if equipped with storage facilities, to catch single events. Logic probes are used to monitor low-speed, digital signals, and to detect short-duration pulses. Both of these instruments can be used to monitor individual signals on a computer bus, but not the traffic on that bus.

Logic-timing analysers; which were designed to assist in fault finding complex, high-speed, logic circuits (Allan 1977, Carver Hill 1974); monitor many signal lines. Readings are triggered by a clock signal which is derived from a bus synchronisation signal, or from a processor clock. Thus, because the analyser is synchronised to the bus, the data read by the analyser is identical to the data latched into the device at the receiving end of the bus. Some logic-timing analysers have been enhanced to indicate if bus activity has occurred between measurements.

Logic-timing analysers met with success as circuit fault-finding tools, and it wasn't long before they were being used to monitor program flow. From this beginning, manufacturers (Hewlett Packard 1978b) developed the logic-state analyser (figure 5.1), claiming it to be a valuable tool for monitoring software, for code optimisation, and for performance analysis. Many of the features included in logic-state analysers were initially developed in the design of customised memory-bus monitors (Fryer 1973).

Performance evaluation research at the University of Wollongong has included using a logic-state analyser as a hardware monitoring-tool. This chapter discusses the analysis of the interrupt-handling routines of the UNIX* operating system, the effectiveness of the logic-state analyser as a performance-measurement tool, and the improvement in performance that resulted from redesigning the common-interrupt program to use architectural features of the Perkin-Elmer computer.

---

* UNIX is a trademark of BELL Laboratories

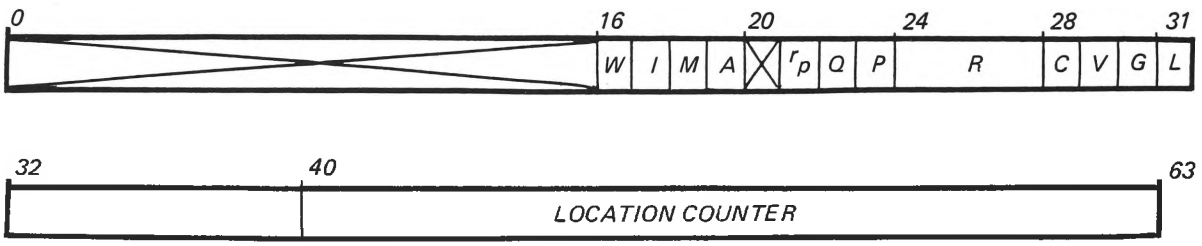
5.1. Performance Measurement

Traditional hardware tools have provided limited information, and, if not designed into the computer hardware, can be difficult to implement. Logic-state analysers are the latest development in the stored-program class of hardware tools. They contain some "intelligence" and have great flexibility. One goal of the work discussed in this chapter was to evaluate the effectiveness of the logic-state analyser as a monitoring tool. An effective tool is one that can be easily attached to a data path in order to collect user-selected information, such as the execution path of a program. Some desirable features are: powerful event filtering, accuracy, ease-of-use, and a range of measurement methodologies that enable data collection at different levels in the object hierarchy.

A second goal was to evaluate the change in performance of interrupt handling, in the UNIX operating system, resulting from modifications to the common interrupt-program. UNIX was moved from Digital Equipment Corporation PDP-11 series of computers to Perkin-Elmer thirty-two bit series machines by Richard Miller (1978) at the University of Wollongong. The Perkin-Elmer version is now available commercially, a world first in portability, motivating evaluation of its performance in an environment it was not designed for.

Evaluation was started by measuring the low-level routines running in the computer, commencing with a completely idle system. When no user programs are active the operating system is occupied solely with housekeeping functions, mainly the time of day clock. Following this, we measured the system while it was executing a single CPU bound program. The philosophy behind this approach is:

- These routines are relatively small, and thus, easy to modify and measure.
- Many of the routines constitute fixed overheads, which have to be taken into account when measuring higher-level behaviour.



Bit 16	Wait state	Bit 21	Enable Relocation/Protection
Bit 17	Enable Immediate Interrupts	Bit 22	Enable System Queue Service
Bit 18	Enable Machine Malfunction Interrupts	Bit 23	Supervisor mode
		Bit 24:27	Register set
Bit 19	Enable Arithmetic Fault Interrupts	Bit 28:31	Condition code

Figure 5.2 Program-Status Word

- Only a small part of the operating system is written in assembler; the majority of the operating system is written in the 'C' programming language. To enable the transfer from PDP 11 to Perkin-Elmer the assembler code had to be rewritten, making it an area of interest. The assembler code includes the the common interrupt-handling-program.

- The heart of any operating system is the scheduler and its associated interrupt-handling routines. It was imperative that these be understood before more extensive measurements were made.

5.2. Interrupt Handling

All processes running under UNIX are started in response to interrupts generated by external events. External events include: user input from a keyboard, character output to a video display, clock ticks, and the completion of disc operations. A common interrupt-program performs actions common to all interrupts, including saving user registers, and then calls the appropriate interrupt-handler. Information required by the handler is passed to it from the common interrupt-program via a standard stack linkage. Thus, interrupt handling software is modularised, and adding new handlers is reasonably simple. When processing is complete the system returns to idle, where it remains until the next interrupt occurs.

A variety of interrupts, all of which can be inhibited by masks in the program-status word (figure 5.2), occur within a Perkin-Elmer 7/32 system. External devices interrupt the processor through the single-level, immediate interrupt. When an interrupt occurs the processor completes the current instruction before servicing the request. The first step, in the hardware interrupt-response-sequence, is to save the current program-status-word. Then the status portion of the program-status word is set up: to select the supervisor register set (the Perkin-Elmer 7/32 has two sets of sixteen, thirty-two bit, general-purpose registers), to disable interrupts, to allow the execution of privileged instructions, and to disable memory relocation and protection. After the interrupt is acknowledged, the address (device number) and status of the interrupting device are loaded into registers. The device number is used to calculate the address in the interrupt-

service-pointer table where the start address of the interrupt handler is located. Finally the start address of the handler is loaded into the location counter and program execution commences.

5.2.1. Clock Interrupt Handling

To illustrate the general principles of interrupt handling, the clock-interrupt handler is included in this discussion. The following description of the clock handler is included to provide the reader with enough background to understand the discussions about performance in later sections. When the clock interrupts the processor (every ten milliseconds), the handler performs the following operations :

- The contents of the memory location addressed by the front-panel switches are displayed on the front-panel, hexadecimal display.
- A function, for example a printer-newline-delay function, can arrange to be called by placing an entry (consisting of an incremental time, a function address, and a parameter) into the callout queue. Every clock tick the incremental time of the first entry in the queue is decremented, and when the time reaches zero the function is executed and the entry removed from the queue.
- User time or system time for the current process is updated, and the total cpu time used by the current process is incremented.

At the end of every second, additional operations are performed by the clock-interrupt handler, including :

- The time of day, measured in seconds since the beginning of the year, is incremented, and the process time of each currently defined processes is updated.
- A flag is set to instruct the common interrupt-program to switch to a higher-priority user-process if one is waiting.

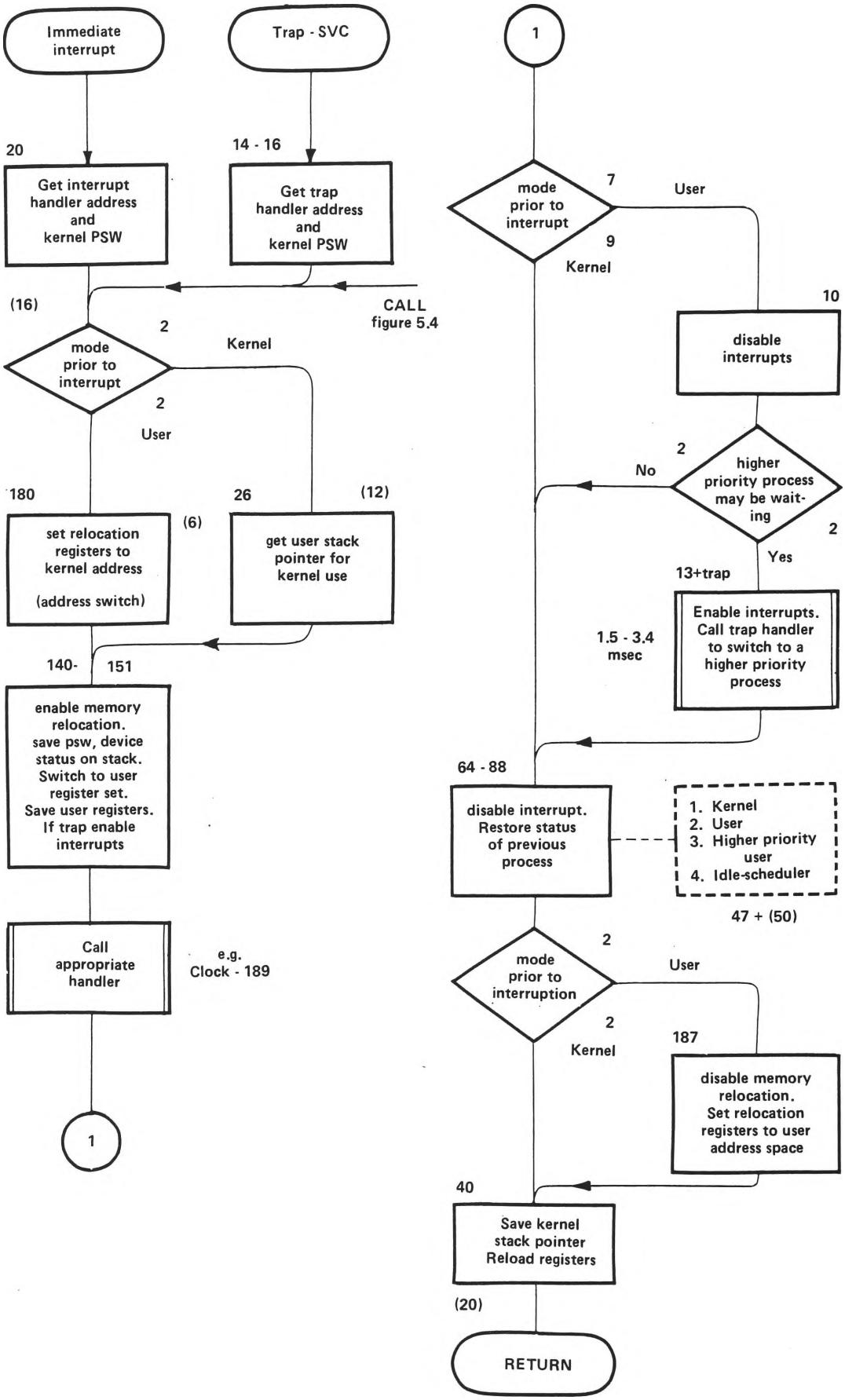


Figure 5.3 Common Interrupt Program. Numbers above boxes are the execution time in microseconds. Numbers in brackets are the execution time of the software monitor that can be added.

- The priority of the current process is reduced, and if it goes below a set level the priority of all currently defined processes is recalculated.
- Any sleeping processes that are ready to wakeup are prepared for waking.
- If the scheduler is waiting to rearrange things, start it running.

The clock routine is a small routine, but it can take longer than the ten milliseconds between clock ticks. So that interrupts are not lost, the clock is able to interrupt itself while doing callouts, and during the once-per-second processing. A flag is set to ensure that these areas of code are not entered when processing the second interrupt.

The regular nature of the clock interrupt, every ten milliseconds, makes the handler relatively easy to monitor. When the computer is idle, only the clock-interrupt-handling routine uses any system resources. Two cleanup routines are started by the clock at regular intervals (0.5 and 5 minutes). Thus, measuring and modelling the clock routine characterises an idle system.

5.3. Common Interrupt Handler

When UNIX was transferred to the Perkin-Elmer computer, the design of the original version was maintained, and the code was written to match the functions of the PDP 11 code as closely as possible. The common interrupt-handling-program, described here and in figure 5.3, was no exception.

When an interrupt occurs, the processor obtains the corresponding vector from the interrupt-service-pointer table, and commences execution at that address. The vector routine obtains the start address of the appropriate interrupt handler, and the desired processor status. Then all interrupts, and software traps (supervisor calls), branch to the common interrupt-handling-program (CALL on figure 5.3).

If the system was previously in user mode, the common interrupt-program loads the memory-relocation registers to switch to the kernels address-space. Then memory relocation and protection are re-enabled. Any information to be passed to the device-interrupt handler is

saved on the stack, and the user register-set is selected. User registers are saved, and if a trap handler, not interrupt handler, is to be called interrupts are enabled.

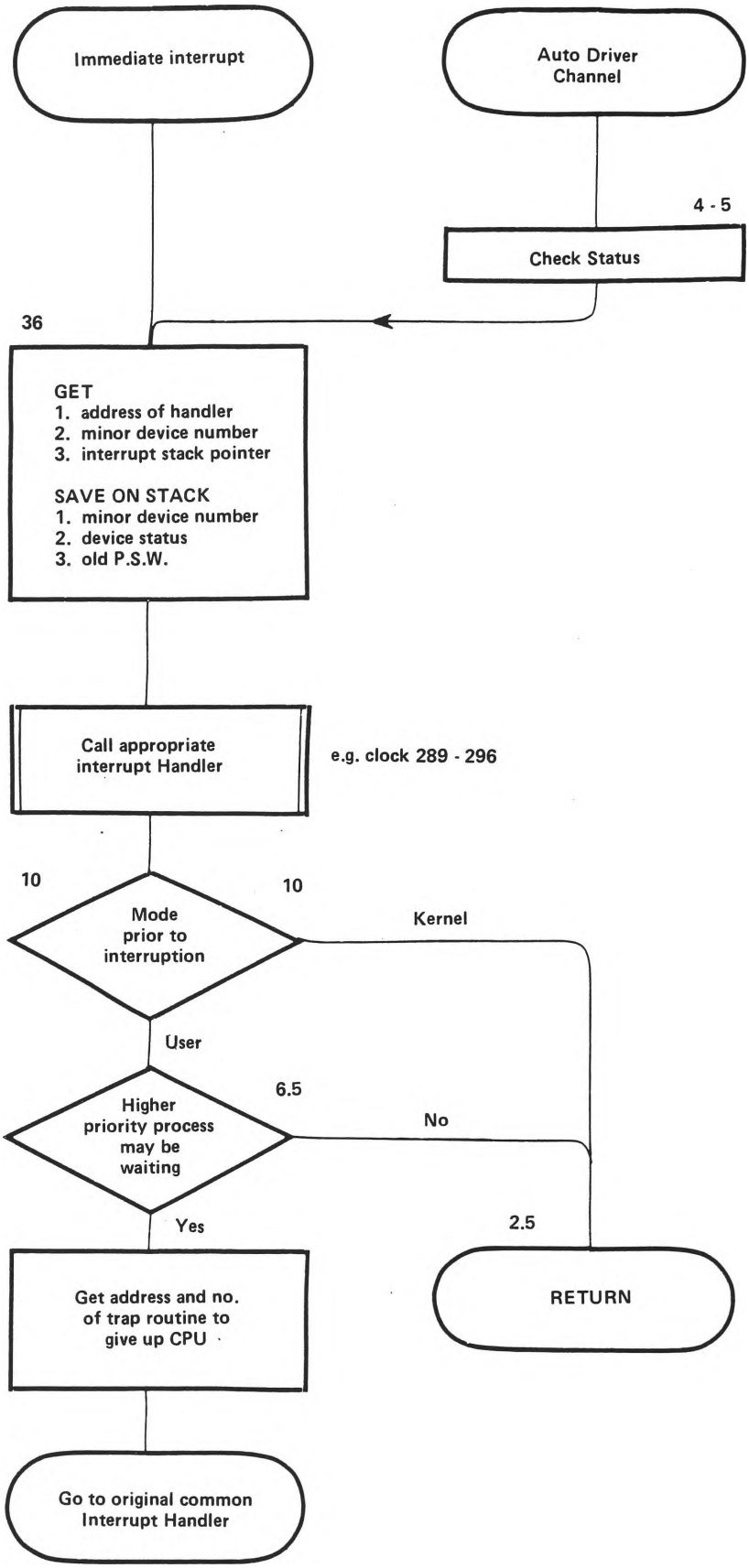
Next the appropriate interrupt handler, or trap handler, is called to carry out the action required to service the interrupt, or trap. On return from the handler, a check is made to see if a higher-priority process may be waiting for service, if the processor was executing a user process prior to the interrupt. Once a second the clock handler sets a flag to force this to happen. If so, the process dispatcher is called to determine if a higher priority process is waiting, and to switch to that process.

Then the common interrupt-program returns to the mode of operation prior to the interrupt, or to a higher-priority, user process.

5.3.1. Modifying the Common Interrupt-Handler

UNIX was written as a multi-tasking operating-system to support a number of programmers in an interactive environment. A common interrupt-handling-system is adequate for this application and has the advantages of modularity and ease of modification mentioned earlier. For real-time applications, the response time of the common interrupt-system is excessive; one of the reasons why UNIX is not a real-time operating system. Real-time systems have to respond rapidly to interrupts because the frequency of the interrupts can be high, and the interrupting device may require fast service. Rapid response can be accomplished by using individual, high-speed, assembler routines, but modularity, ease of modification, and portability may be sacrificed.

A Perkin-Elmer computer, running Unix, was linked to a Univac main-frame computer, running Exec 8, for remote batch job submission. A synchronous RS232C link is used, and data is transmitted one line at a time. Synchronism is established at the start of the first character, and then the whole line is transmitted as a continuous stream of bits. An interrupt is generated at the end of every character - every eight bits . An eighty character line, transmitted at 4800 baud, generates 80 interrupts, 1.667 milliseconds apart, during the 134 milliseconds it takes



CALL - figure 5.3

Figure 5.4 Modified Common Interrupt Program

to complete the transmission. The data transfer rate is effectively half the transmission rate, because of the cumbersome design of the Univac protocol. Protocol handling requires complex, time-consuming software, but the high interrupt-rate requires fast, interrupt-handling software.

To provide a usable link, the common interrupt-handling-program was modified to reduce interrupt handling-time in most situations (figure 5.4). This reduction was achieved by making use of architectural features specific to the Perkin-Elmer range of thirty-two bit computers. Trap handling was not modified. Time is saved in the following ways:

- An address switch is no longer done. Memory relocation is inhibited while servicing the interrupt, and thus, program addresses are physical addresses.
- A special stack was provided for the interrupt routines to use.
- All interrupt processing is done using the supervisor register-set, and consequently there is no need to save and restore the user register-set, or to pass data from one register-set to the other through memory.

An additional architectural feature was utilised to make system reconfiguration easier. Interrupts no longer vector to separate locations, to obtain handler information, before branching to the common program. Handler information is placed in tables, and interrupts from all devices, except the auto-drive channel, vector to one address. The physical-device number, which has already been placed in a register by the processor, is used to index into the tables to obtain the address of the handler, and the minor-device number.

In the case of an interrupt to a user process, where the flag is set to indicate that a higher-priority process may be waiting, the new program branches to the entry point of the original common-interrupt-program (CALL on figure 5.3). Otherwise a return is made to the state prior to the interrupt: user process, kernel process, or idle.

Exception Type	Previous Mode	Return Mode	Execution Time	
			Before	After
Interrupt	Kernel	Kernel	303-336	49
"	User	User	654-687	55
"	User	Higher priority user	667-700	689
	Idle	Idle	350	96
Trap	Kernel	Kernel	297-332	no change
"	User	User	648-683	"
"	User	Higher priority user	661-693	"

Table 5.1 Execution Time of the Common-Interrupt Program - before and after the program modifications - (all times in microseconds)

5.4. Performance Improvements

The execution of the common interrupt-program was studied in detail with the logic-state analyser. Execution times for each section of the program, measured before and after modification, are given in figures 5.3 and 5.4. In two sections of the flow chart, in figure 5.3, a time range is shown. This time range is a consequence of variations in the time taken by the processor to calculate the effective addresses of indexed instructions.

Times in brackets are the execution times of a software monitor which can be added to the program. This monitor reads a precision-interval clock to measure system time, user time, and idle time to the nearest millisecond. For every interrupt to a user process this tool costs 42 micro-seconds, and for an interrupt to a kernel process it costs 48 micro-seconds. When a return is made from the common-interrupt program to idle the monitor costs an extra 50 micro-seconds. System accounting-programs read the counters maintained by the software monitor, and record their contents.

If the machine is idle when interrupted, the interrupt is serviced, and then the scheduler entered, where, depending upon the action of the interrupt handler, either a process is started or the system returns to idle. A return to idle via the scheduler takes 47 micro-seconds.

Execution times for the common-interrupt program, for various processor states, are summarised in table 5.1. In the initial program, interrupts to user processes are more expensive than interrupts to kernel processes (654 micro-seconds compared to 303 micro-seconds), because when servicing an interrupt to a user process the common-interrupt program must switch from the user address-space to the kernel address-space before calling the interrupt handler, and switch back again afterwards. Loading the memory-relocation registers in the memory-access-controller hardware from a table in memory, to accomplish an address switch, takes 174 micro-seconds. Removing the need to switch address-space accounts for 60 per cent of the savings in the modified program.

Performance improvements arising from modifying the common interrupt-program are:

Clock Function	Execution Time	
	Before	After
Clock tick during a kernel process	492	341
Clock tick during a user process	843-849	347
Clock tick during a user process, and return to a higher priority process	3,031	3,151
Clock tick during a user process with 1 callout - printer new line delay	1,970	1,475
1 second period, kernel process, and no callouts	5,784	5,634
1 second period, user process, and callouts pending	6,517	6,022

Table 5.2 Representative Times for Clock Handling - before and after program modifications - (all times in microseconds)

	Kernel Process		User Process	
	Before	After	Before	After
Common Interrupt Program	303	49	654	55
Clock Handler	189	292	189	292
Total	492	341	843	347
Time Saved	150	150	495	495
Saving in cpu time		1.5 %		4.95 %

Table 5.3 Comparison of Clock-Handler Execution-Times - before and after program modification - (all times in microseconds)

- a reduction in the execution time of the common interrupt-program from 654 to 55 micro-seconds for interrupts to user processes, and
- a reduction in the execution time of the common interrupt-program from 303 to 49 micro-seconds for interrupts to kernel processes.

The time taken to handle traps has not altered, and the time taken to handle an interrupt to a user-process which returns to a higher-priority user-process hasn't changed, because the modified common-interrupt-program calls the original program to achieve the address switch. For the majority of interrupts, processing time has been significantly reduced. Adding the software monitor to the modified program increases the execution time by 40 micro-seconds, making it an expensive tool.

The total time taken to handle an interrupt is the sum of the execution time of the common interrupt-program plus the execution time of the appropriate interrupt-handling-routine. The latter time varies from handler to handler, and varies with processor status. When the system is idle, the time taken to service a clock tick, prior to the modifications, was 541 - 548 microseconds. This time was made up of: common interrupt-program, 303 microseconds; clock handling, 188-195 microseconds; plus return to idle via the scheduler, 47 microseconds. If the software monitor was included an additional 92 microseconds was used. When the processor was executing a user process, servicing a simple clock tick took even longer, 843 - 849 microseconds (table 5.2).

The clock handler is the only handler which had to be modified to compensate for memory relocation being turned off. In order to access user-area structures the clock handler must calculate the physical address of the user stack from the virtual address and the kernel-segmentation table. Clock handler time was increased from 189 to 292 microseconds (table 5.3), as a result of this modification, negating some of the savings in the common interrupt handler.

For a clock tick during a user process the clock service time has been reduced from 843 to 347 micro-seconds. Clock service time during a kernel process has been reduced from 491 to

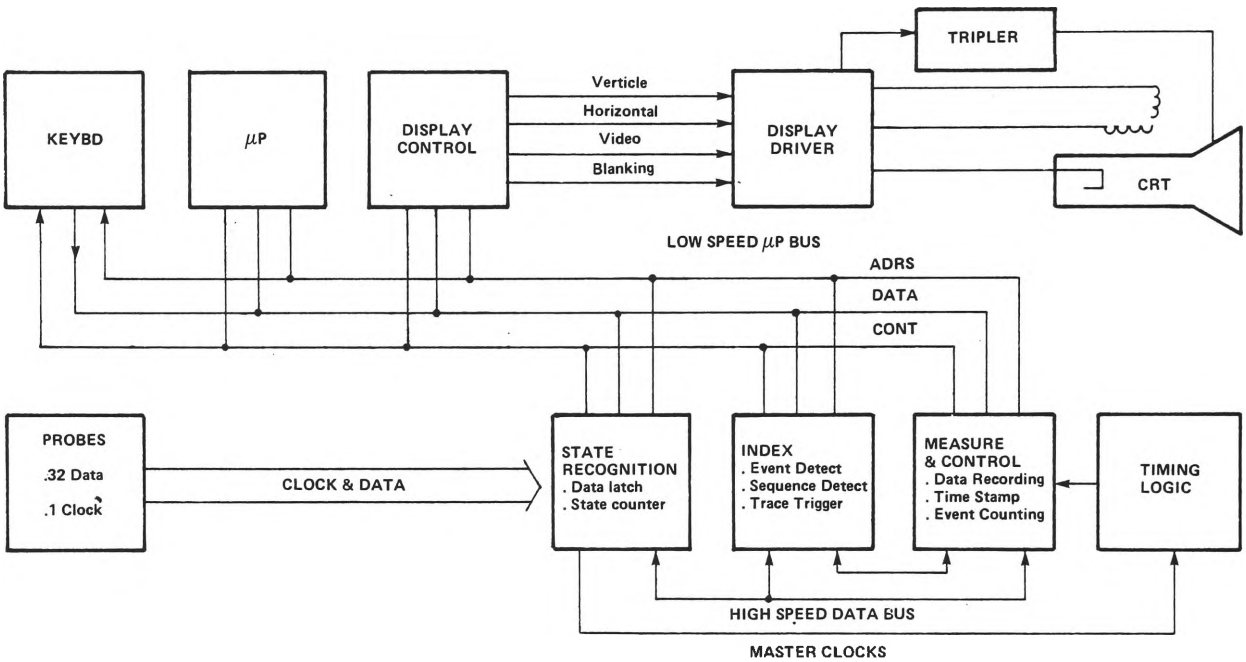


Figure 5.5 1610A Logic-State Analyser Block Diagram

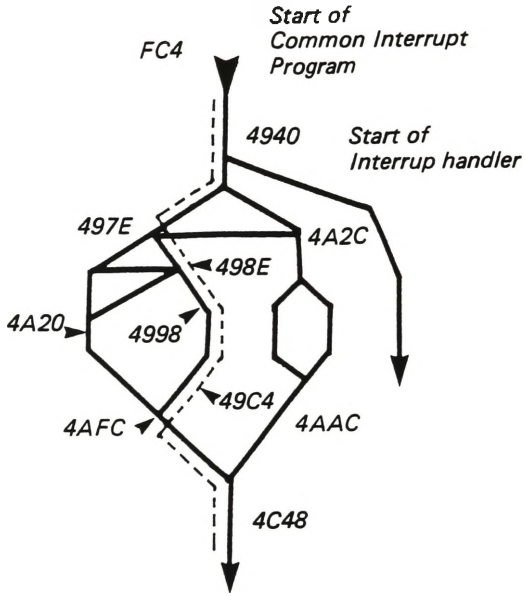


Figure 5.6 Complex network branching requires high-level, sequential trigger capability. e.g. Trace states from 498E only after dashed path is executed.

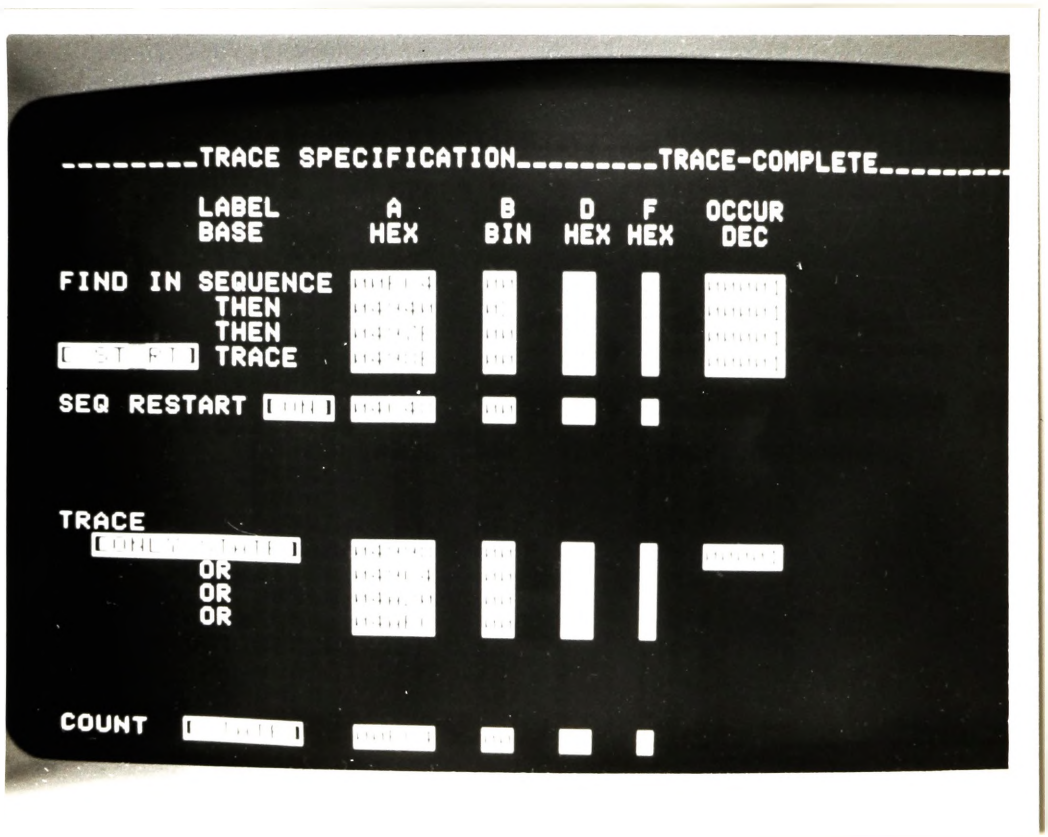


Figure 5.7 Event Filter set up to trace the states after address 498E in the program graphed in Figure 5.6.

341 micro-seconds. This improvement in clock handling represents a **saving of 5 per cent of absolute CPU time** during user processes and 1.5 per cent of CPU time during kernel processes.

Time saved during the processing of interrupts from other devices is equal to the reduction in execution time of the common-interrupt program. For a synchronous link running at 4800 baud, which will interrupt every 1.667 milliseconds, **the saving is 36 per cent of absolute CPU time,** for transmissions during user processes. For asynchronous devices, the saving depends upon the frequency of the interrupts.

5.5. Logic State Analyser

The logic-state analyser (figure 5.5) can monitor thirty-two digital-signals simultaneously, at a maximum sampling rate of ten megahertz (Hewlett Packard 1978a). It can store sixty-four readings, triggered by an external clock, and display any sequential group of twenty-four of these. The analyser can be set to trigger on a sequence of data-patterns, and once triggered it will record the occurrence of specified data-patterns (states) until its memory is full.

Unlike traditional hardware monitors, the logic-state analyser doesn't have a patch board at the front end for manual selection of event filtering. Instead, an interactive video display is used: to split signals into logical groupings, to define triggering sequences, to specify states to be traced, and to enable counting functions. The microprocessor, internal to the analyser, sets electronic switches and counters to perform the tasks normally done by the patch board.

Event filtering is powerful enough to allow a path to be traced through a complex branching network, such as that in figure 5.6, before any states are recorded. Then it will trace all states, or only selected ones, as desired, and record the time between states, or the frequency of occurrence of a specified state. An event filter set up to monitor the network in figure 5.6 is shown in figure 5.7. The input signals have been split into groups: A is the address bus, D is the least-significant byte of the data bus, and B is status information. This event filter specifies a series of states that must occur in sequence before tracing starts, and specifies four states to be traced. As only four states are traced the analyser will record several passes through the

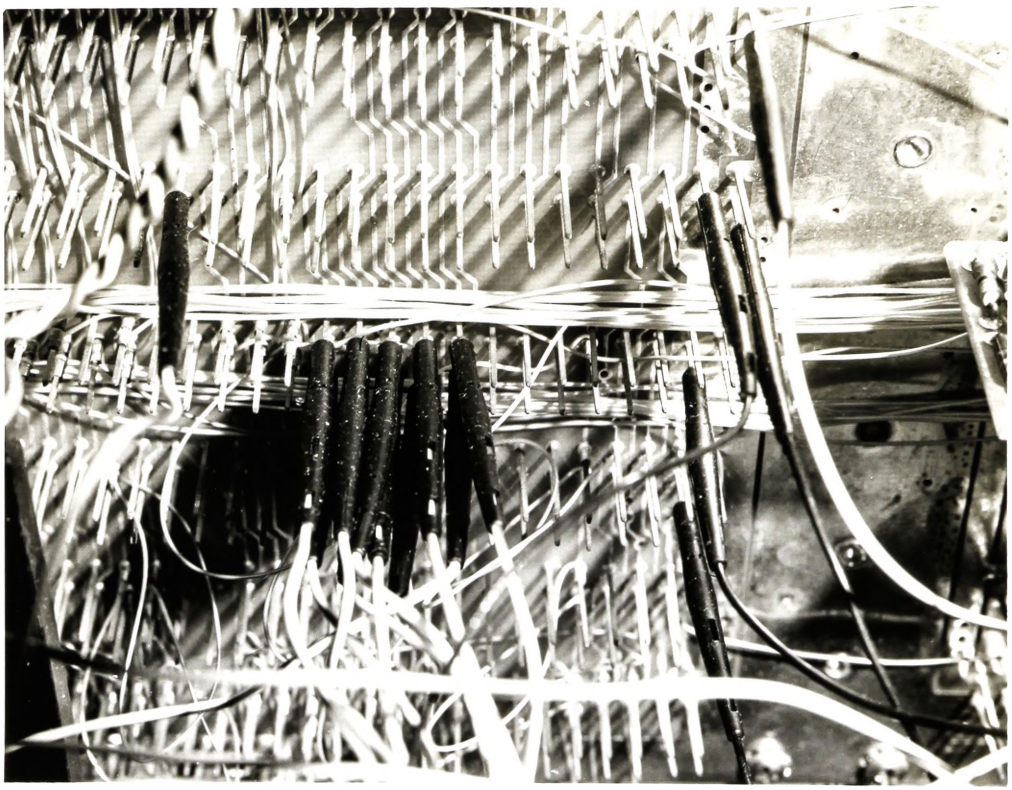


Figure 5.8 Connection of the Analyser to the Computer's Back-Plane.

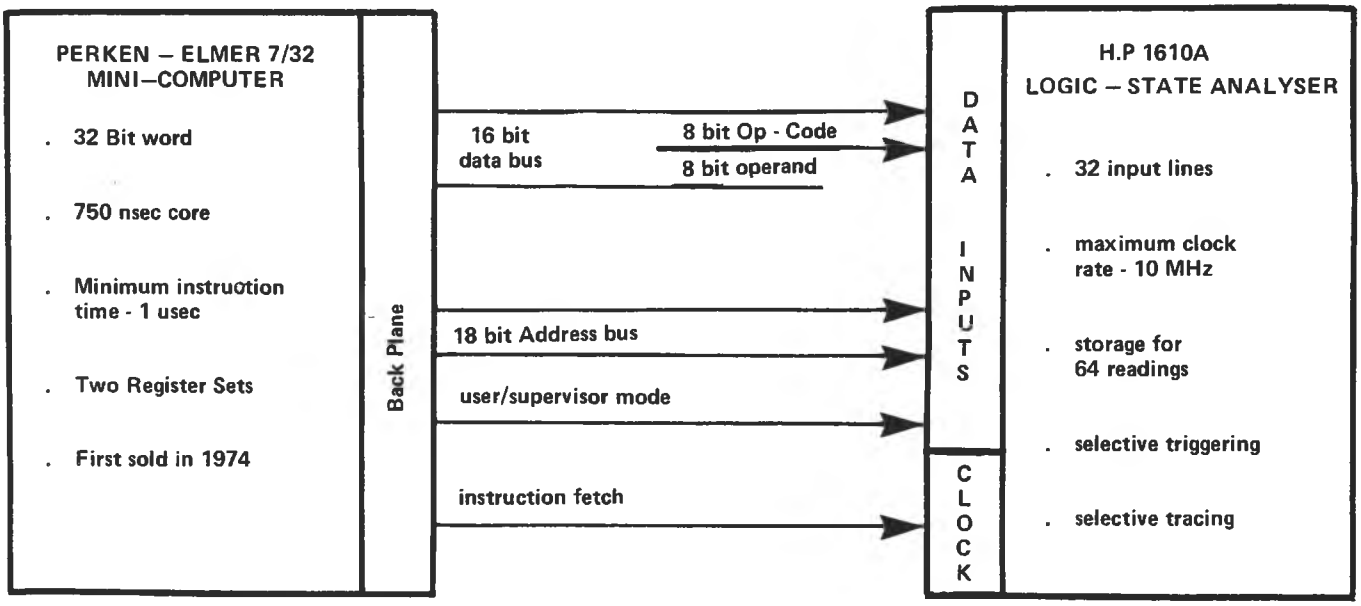


Figure 5.9 Back-Plane to Analyser Signal Connections

code, but only trigger once, so subsequent passes may not necessarily follow the same path. To indicate path variation, a state (4A20) from the alternate path is also traced. A measure of interrupt activity during the measurement period is obtained by counting the entries to the common interrupt-program (state FC4).

The analyser is connected directly to the backplane of a Perkin-Elmer 7/32 processor (figure 5.8). Insulated connectors, supplied with it, push onto wire wrap pins, making a convenient and safe connection. Eight bits of the data bus, sufficient to read the operation code, the eighteen-bit address bus and some status lines (figure 5.9) are connected to the analyser. Data-bus information can be compared to a machine-code listing of the program being monitored to establish confidence in the analysers operation.

To synchronise the analyser to the correct phase of the minicomputer's memory-access-cycle, a clock signal, derived from the bus-control logic, must be connected to the analyser. The signal chosen is activated when an instruction-fetch is initiated by the microcode. A consequence of selecting this signal is that other memory cycles, such as data fetch, are not seen by the analyser, effectively filtering out much bus activity. When analysing the common-interrupt program we were solely interested in tracing the execution path of the program and so this filtering was desirable.

5.5.1. Measurement Methodology

At the machine-code level, the execution path can be traced by triggering on the start address of the program and recording the first sixty-four instruction addresses. By repeatedly updating the trigger address, to the last address recorded on the previous trace, the execution of the whole program can be recorded. If several recordings are taken, at each trigger point, and compared, most program branching points can be found. Then, the recordings are compared to a machine code listing to extract a flow chart of the program, and to detect any missed branches. Finally, the execution time for each section of the code is calculated from the timing data recorded by the analyser.

With large, complex programs the above method can become tedious, and may provide more detail than required. Important information, such as execution path and execution time for each path through a program, can be obtained using a simpler approach. In the analysis of the clock handler, the analyser was set to trigger on a clock interrupt, and then, to trace the clock-routine entry address, exit address, and an address in each program branch of interest. Execution path is indicated by the occurrence of a branch address in the recording, and execution time is the time period between the occurrence of entry and exit addresses. In this way, the program-level object was decomposed into block-structure level modules.

The clock routine is written in the 'C' programming language. The 'C' compiler produces assembler mnemonics, which are assembled to machine code, and an assembler listing is produced. This listing was used in conjunction with symbolic debug to find program entry, branch and exit addresses for state analysis. If the entry and exit addresses of the called routine are not known the addresses of the call and return statements of the calling routine can be used.

Modifications to the common interrupt-handler created some problems for state analysis. In the modified program, all interrupts vector to the same location, making it difficult to distinguish one interrupt from another. Interrupt-specific information is contained in memory locations and registers. This problem was overcome by setting the analyser to trigger on the entry point of the interrupt handler rather than on the interrupt-vector address.

If processor-status information is available to the analyser even more powerful methodologies can be devised, allowing the system object to be decomposed into task-level modules. When an interrupt occurs, the processor switches from user to supervisor mode by setting a flag in the program-status word (figure 5.2). This signal is available on the back-plane and was connected to the logic-state analyser (figure 5.9).

The total time taken to handle an interrupt to a user program can be measured by triggering the analyser on an address unique to the desired handler, and tracing only user-mode instructions. The time difference between the user-mode instructions, immediately before and

after the interrupt, is the time taken to handle the interrupt. Again, specific execution paths can be monitored by tracing addresses in branches of the program. To simplify initial measurement, only one user process was running on the system; a "BASIC" program in an infinite loop.

5.5.2. Limitations

A lot of useful information is provided by the preceding measurements (table 5.2), but to model the clock routine completely two areas need further investigation: the callout section, and the process-table update section. Execution time for the callout section depends upon the number of callouts, and the functions called. For example, a function called to provide newline delay on a line printer takes 1,068 microseconds. Process-table-update execution-time depends upon the number of current processes, the number of processes ready to wake up, and the priority of the current process. If the state analyser could read memory locations this stimulus information could be determined during the trace, thereby simplifying measurements. At present, this stimulus information is determined either by simulating conditions or by using software tools to determine the current system state.

Hardware monitoring-tools are inherently limited to measuring information that can be interpreted at the hardware level without knowledge of software activities. In the case of the logic-state analyser, there is a solution if the information required is stored in fixed, known, memory locations, and if the memory contents are updated regularly. The solution is to use the memory-cycle as an analyser-clock signal, and to connect memory-access status-information, such as instruction fetch, and data write, to analyser inputs. Then the analyser can be set to trace memory-data writes to a specified memory location, to obtain program specific information, in addition to tracing instruction fetches.

Whether the above can be done or not, depends upon the availability of the desired memory-access status-signals, the width of the analyser, and the triggering capabilities of the analyser. The analyser used in this study is thirty-two bits wide, has one clock input, and only eight registers available for specifying triggering sequences and states to be traced. If a second

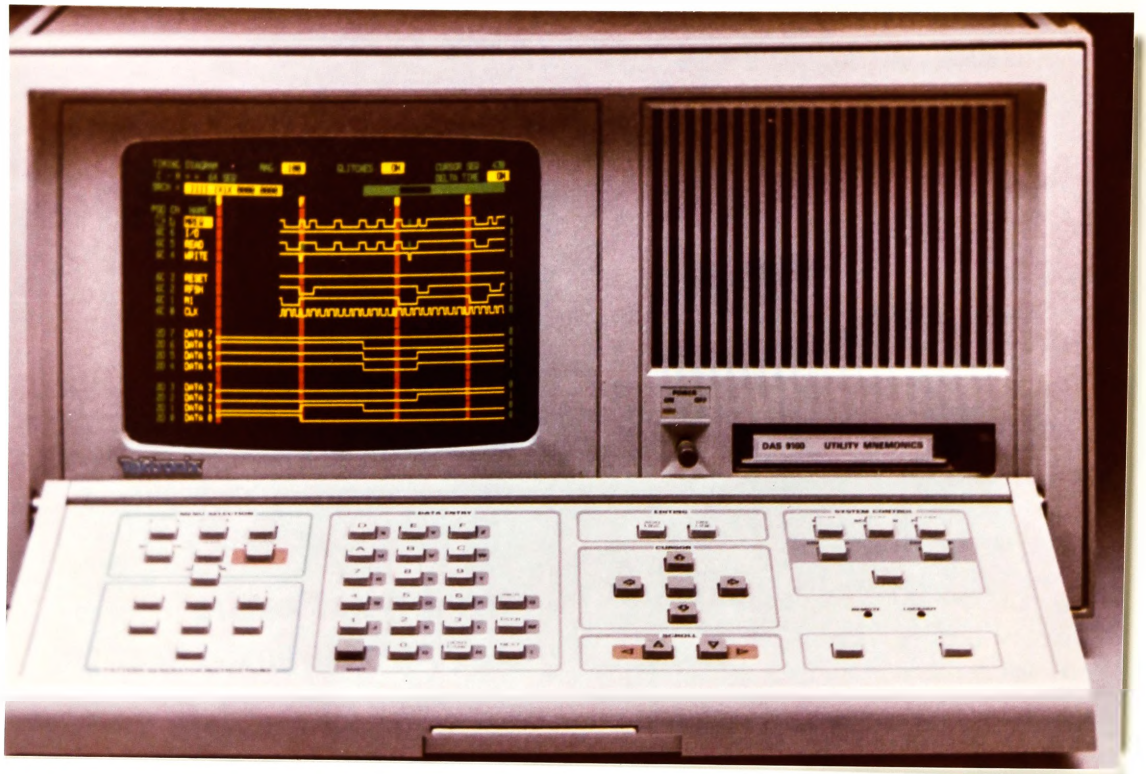


Figure 5.10 Colour Display on the Tektronix 9129 Digital Analysis System.

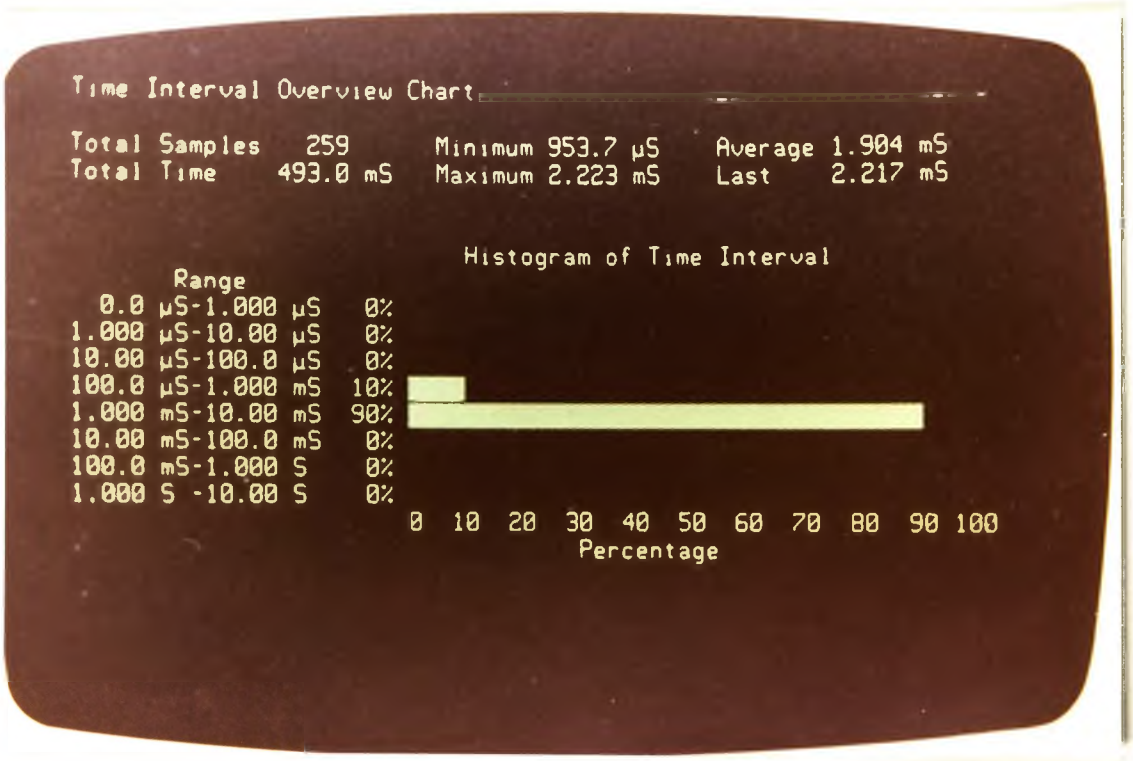


Figure 5.11 A time histogram display on the HP 1630 Logic Analyser.

clock input was available, a dual clocking system - one for data write, and one for instruction fetch - could be used as an alternative to the above solution, freeing up some of the analyser inputs.

5.6. Newer Logic-State Analysers

The logic-state analyser used in this research is now obsolete. A range of new analysers, from a number of manufacturers, have recently been released onto the market (Comerford 1981, Brampton 1982, Guteri 1982, Corson 1983). The principle of operation is the same as in earlier analysers, however, advances in technology have enabled the designers to increase the power of the tools considerably. All have increased resolution, more inputs, more storage, and all combine the facilities of both state and timing analysis into the one instrument. In addition, event filtering facilities have been enhanced and some analysers can handle multiple clocks. Optional plug-in modules extend the range of the basic instrument and provide new facilities, for example - a programable pattern generator for stimulus generation.

Considerable emphasis has been placed on improving the user interface and data presentation features. Tektronix (figure 5.10) use a colour display to highlight areas of interest. Hewlett-Packard (figure 5.11) have included a histogram display, and enable the user to label measured signals.

A major application of logic analysers is the analysis of microprocessor software. One of the problems of measuring most microprocessors is that no signals are available to separate instruction-fetch memory-cycles from other memory cycles. To measure the execution of programs by the microprocessor, the microprocessor is either replaced by an emulator or a state machine is used to detect instruction-fetch memory-cycles. These facilities are built into microprocessor specific personality modules, to enable easy connection to all bus signals. If an emulator is used, the microprocessor is removed from the target system and an interface connector is plugged into the dual-in-line base. If a personality module is used, an IC clip is used to connect the microprocessor to the personality module.

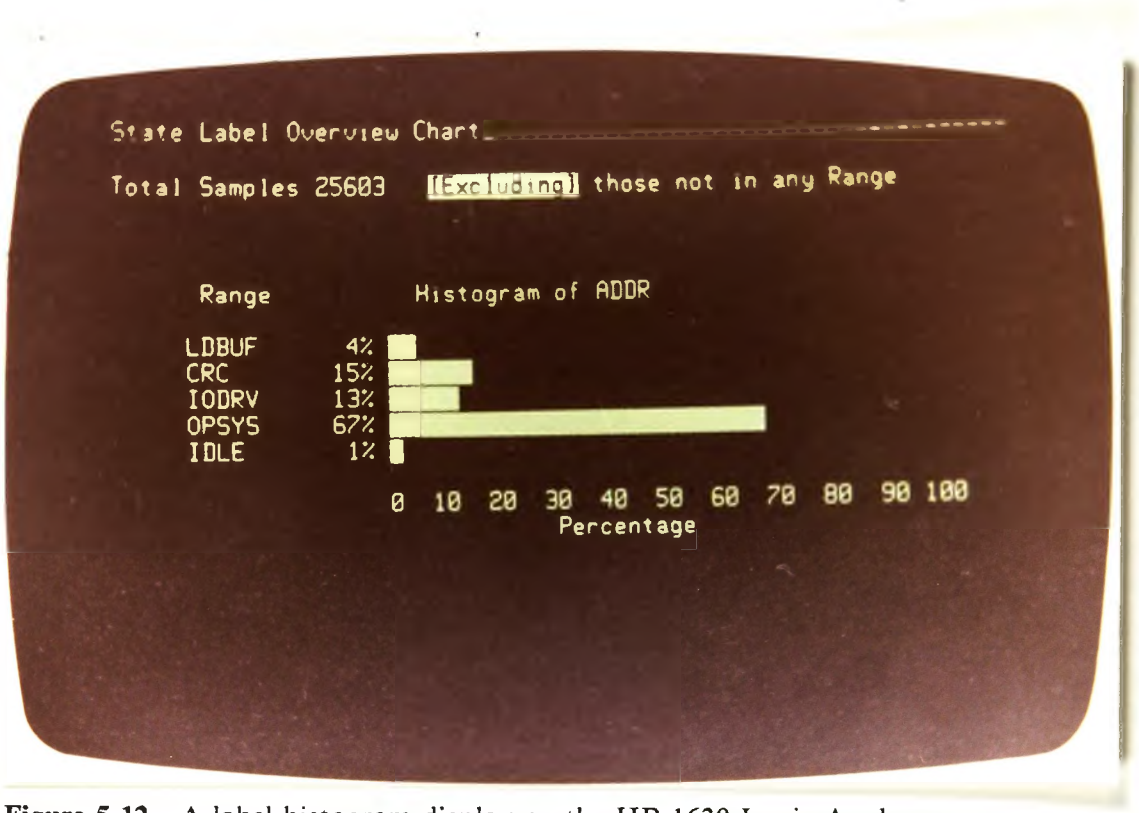


Figure 5.12 A label histogram display on the HP 1630 Logic Analyser.

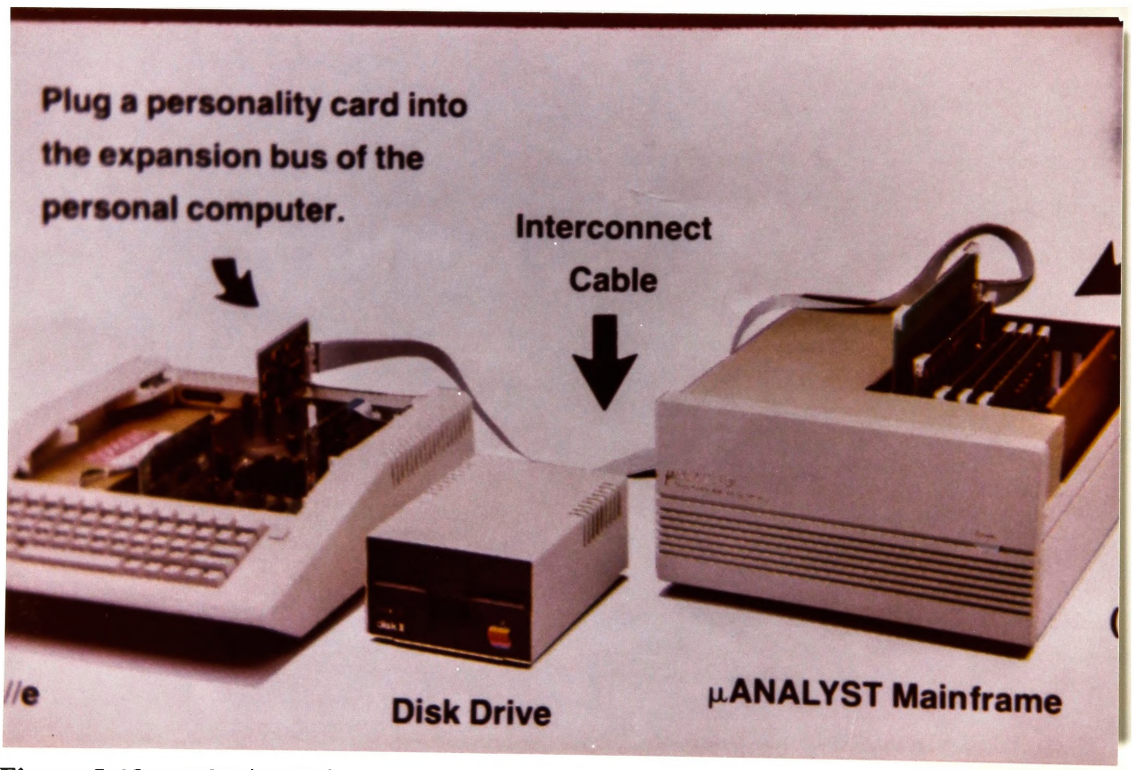


Figure 5.13 A logic analyser controlled by a personal computer.

Software running on the logic analyser, using either an emulator or the state information produced by the state machine in the personality module, maps the operation codes into mnemonics. These mnemonics are displayed together with the address of the instructions. Memory activity between instruction fetches is also indicated on the display. Thus, very powerful assembler-debugging tools are available.

Hewlett-Packard (Corson 1983) have also included some performance-evaluation tools in their logic analyser. The time period between two user-defined events (module execution-time) and the number of times that path was taken can be measured, and displayed as a path-utilization histogram (figure 5.11). A second display (figure 5.12) presents a histogram of address space usage. By defining the address space in terms of program labels, the analyser lets the user quickly identify the location of maximum system activity. These tools are both useful and powerful, but require an address map of the program being analysed, limiting their usefulness in high-level language and dynamic-memory-allocation applications.

Of more interest to performance analysts is the trend toward user programable analysers. Such analysers are equal in power to many hardware monitors, and enable users to write their own analysis software. Possibly the most exciting instrument is the μAnalyst from Northwest Instrument Systems (figure 5.13). The μAnalyst contains the probes, event filtering circuits, and storage of a traditional logic-state analyser. Computing power is provided by a host personal computer (Apple II, IBM, Compaq), into which the μAnalyst plugs as a peripheral. The μAnalyst is roughly one third the cost of an equivalent stand alone instrument, plus the cost of the personal computer.

Data collection, event filtering, data analysis, and data display are all under the control of software running on the personal computer. Each probe card can connect to sixteen data signals (up to 5 cards can be used), with a resolution of 10 MHz, and can store up to 4096 measurements.

Measurements are recorded directly into the monitoring-processor's memory, eliminating

the data link which is needed to connect a stand-alone state analyser to a controlling computer. Thus, the data transfer overhead is eliminated, and analysis software can analyse the data as soon as the measurements are made. With a stand-alone analyser the data is not transfered to the controlling processor until the measurement buffer is full. Thus, a delay exists between measurement and analysis. Also, data recording stops until data transfer is complete, introducing discontinuities into the trace.

Some measurement software is provided with the system, and additional measurement software can be written by the user as needed. This instrument has the potential of providing cheap, powerful, user programable, performance measurement facilities to a larger group of computer programmers than ever before.

5.7. Conclusion

The logic-state analyser has proved to be a useful performance-evaluation tool, because of its ability to measure object execution paths, and execution times.

Objects can be examined in great detail - down to machine-code level. Program-level objects can be monitored by tracing program entry points, exit points, and branch paths. Execution time of interrupt handlers can be measured by measuring the time spent in supervisor mode servicing the request. State analysis is a powerful and accurate measuring tool which does not interfere with the software it is monitoring. Limitations which have shown up in the logic-state analyser as a performance evaluation tool are typical of hardware tools.

Code written to be portable and easily modifiable often does not make use of specific features of the target architecture. Modifying code to use these architectural features improves performance at the cost of portability. Performance improvements, resulting from modifying the common-interrupt program to take advantage of processor architectural features, include a reduction in the execution time of the common-interrupt program from 654 to 55 microseconds, when servicing interrupts to user processes. For a synchronous link running at 4800 band, this reduction in execution time represents a saving of 36 per cent of absolute CPU time, during

transmissions.

A surprising amount of CPU resources, **3.41 per cent of CPU time minimum**, are consumed by clock handling. Thus, the selection of the clock interrupt-rate is an important consideration in system design.

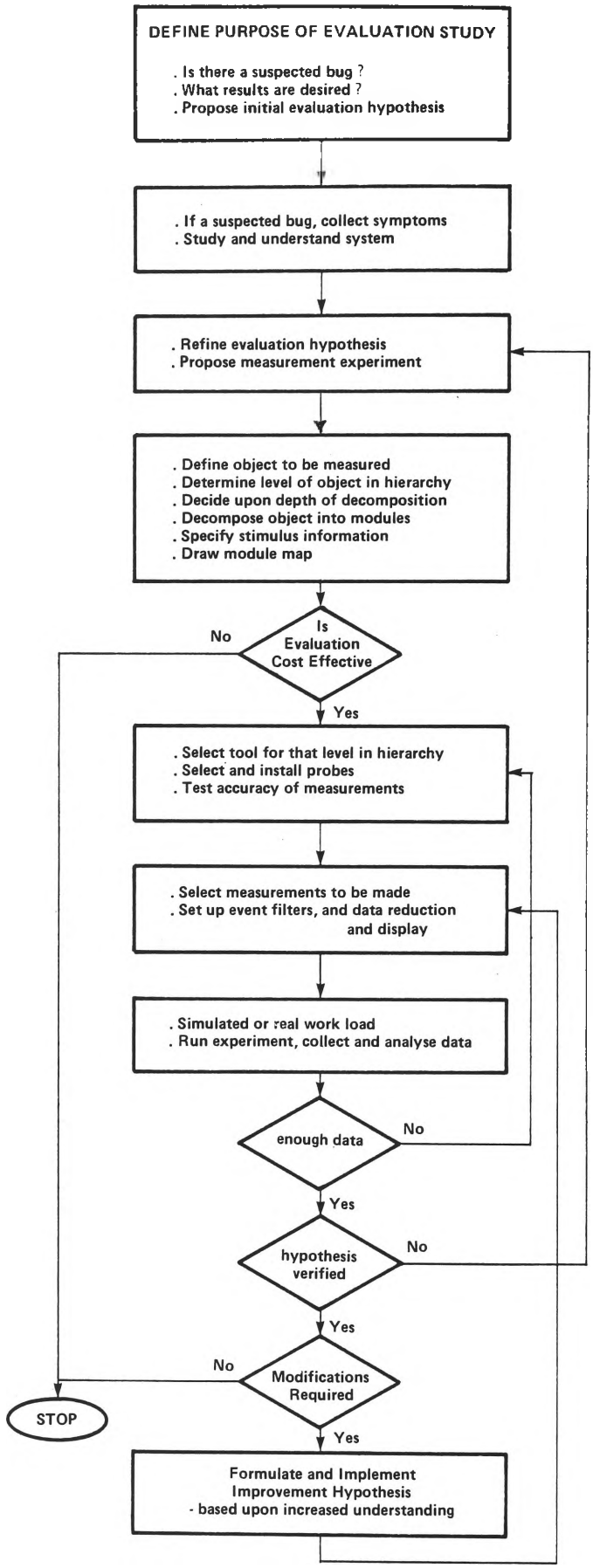


Figure 6.1 Measurement Methodology

6. Measurement Methodology and Tool Design

In the introduction (section 1.4), the need for a measurement methodology based upon the scientific method was discussed. One of the corollaries (section 2.13) to the formulation of performance measurement is that the formulation provides a general, overall context within which measurement and evaluation can take place. In this chapter, a measurement methodology (figure 6.1) is developed from the formulation. Also, the design of a hybrid measurement tool is discussed in terms of the formulation.

In the formulation of performance measurement, the set of measurements which can be made on an object have been defined. Which measures are to be made, during a performance evaluation study, depends upon the hypothesis to be tested. However, a comprehensive measurement tool should include all the possible measures, and enable the analyst to select those measures desired for hypothesis testing.

The analyst should first define clearly the purpose of the evaluation study: what are the expected results, is there a system bug to be found, etc. From this information, an initial hypothesis can be proposed, and a measurement experiment formulated. Then, in the case of a bug, all the symptoms should be collected. Once the object to be measured has been delineated, it should be studied until its operation (or expected operation) is understood. Now the hypothesis can be refined.

At this stage, the analyst should know enough about the system to define clearly the object to be studied, the level of the object in the measurement hierarchy, the depth of decomposition required to give the desired measurement resolution, and the decomposition of the object into modules. Once the modules are defined, a method of instrumentation, suitable for that level of the object hierarchy, can be selected. Probes have to be inserted into the system to detect the start and termination of modules, and to record stimulus information. If the decomposition is more that one level deep, then the probes must also detect the start and termination

of objects in the hierarchy. A module map; which includes: module name, module identifier, module function, and stimulus variables; is constructed as the system is instrumented.

The amount of instrumentation is determined by the purpose of the measurement. In the section 4.5.1, we saw how Svobodova (1973a) neatly decomposed a system to the task level by inserting one probe into the scheduler, demonstrating the benefits of understanding the system. The same results could have been obtained with much more extensive instrumentation, causing considerable interference to the operation of the system. To decompose the system further would have required extensive instrumentation, but that was not needed to test her hypothesis.

In the measurement methodology developed for using the logic-state analyser as a hardware tool (section 5.5.1), a number of decompositions were used. The finest was the decomposition of the program object into individual machine code instructions. A higher-level decomposition involved decomposing the interrupt handling system into a number of routines. Monitoring the processor-mode status-flag enabled the system to be decomposed into a kernel-mode module and a user-mode module. Individual process modules could then be monitored by selecting suitable experiments.

Two other case studies are included in later chapters of this dissertation. In chapter 7, the decomposition of a Pascal-program object into block-structure-level modules, by the insertion of software probes into the object, is discussed. In chapter 8, a small system which has had instrumentation integrated in at design time is described.

Once the system is instrumented, tests have to be run to verify the accuracy of the instrument. The analyst can then select the measurements to be made, run the measurement experiment, and analyse the data. Depending upon the results, the system may be modified, new hypothesis proposed, old hypothesis refined and the experiment re-run, or the measurements terminated.

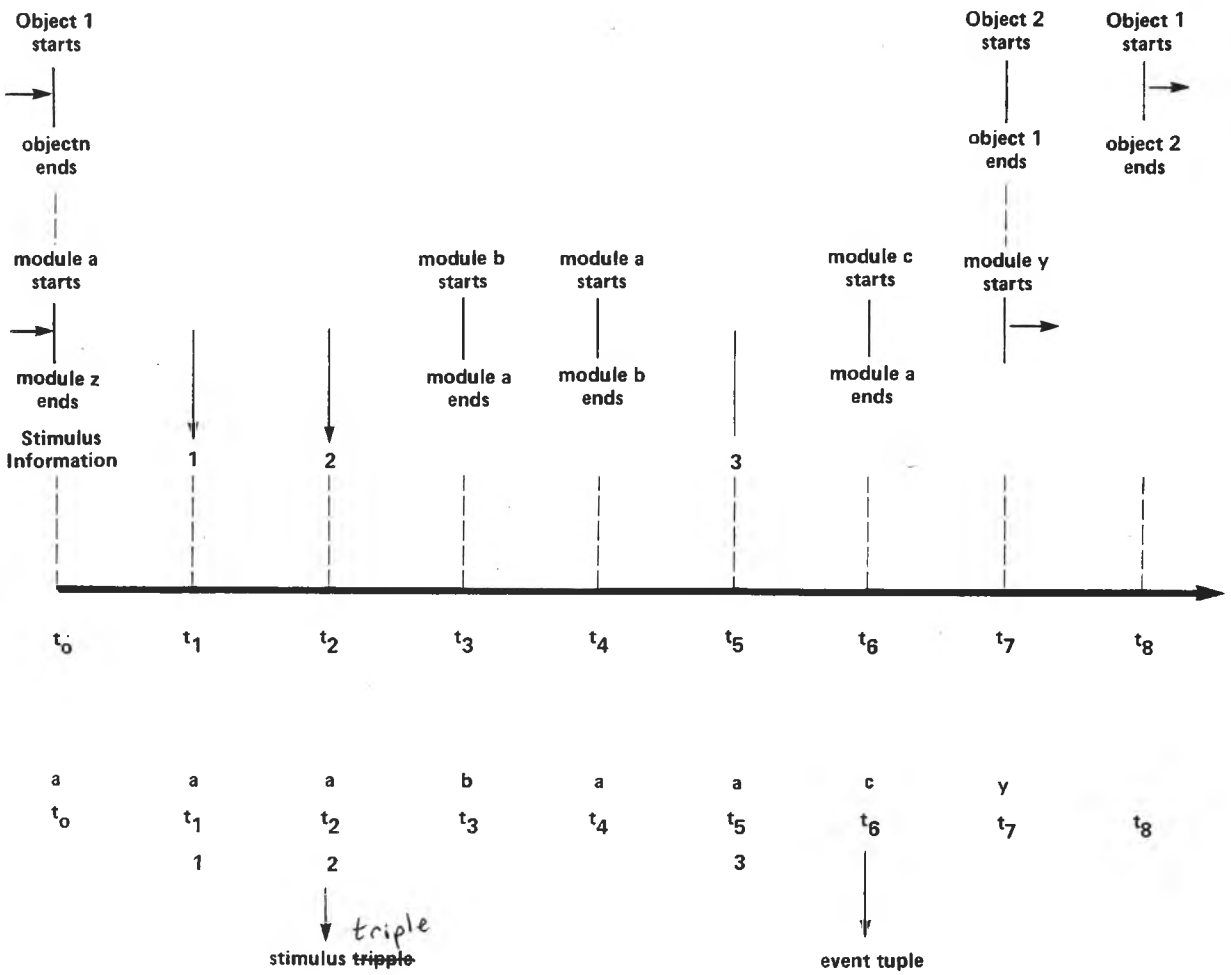


Figure 6.2 Event Trace - showing the sequence in which events are recorded.
1. The end of one object and the start of the next may be seen as two views of the same event.
2. The start of an object and the start of the first module in the object are also two views of the same event.

6.1. Measurement of Objects

6.1.1. Event Detection

A monitoring tool should record an event trace of modules, and associated stimulus information (figure 6.2). The event trace is a sequence of tuples containing a module identifier and the time, relative to the start of the measurement period, at which the module commenced execution. Stimulus information is recorded in the event stream as a triple: current module identifier, time relative to start of measurement period, one (or more) item(s) of stimulus information.

The object must be instrumented to detect the following events:

- start of object execution,
- pause in object execution (if the module is interruptable),
- resumption of object execution, and
- termination of object execution.

These events must be detectable both for the object and for all the modules into which it is decomposed during the measurement experiments. To instrument a system to indicate clearly each of these events is a fairly complex task.

Usually, a significant reduction in the number of event tuples (and hence the number of probes and the interference) is achieved by one tuple signifying more than one event. The termination of one object, or module, and the start of next occur at the same instant and can be recorded as one event tuple rather than two. The start of an object and the start of a module always occur at the same time, hence they can be recorded as one tuple. A pause in a module and the start of another occur at the same timer, as does the end of the latter and the resumption of the former.

This data compression causes an apparent loss of information which must be reconstructed either by the analysis section of the tool or by extending the information in the event

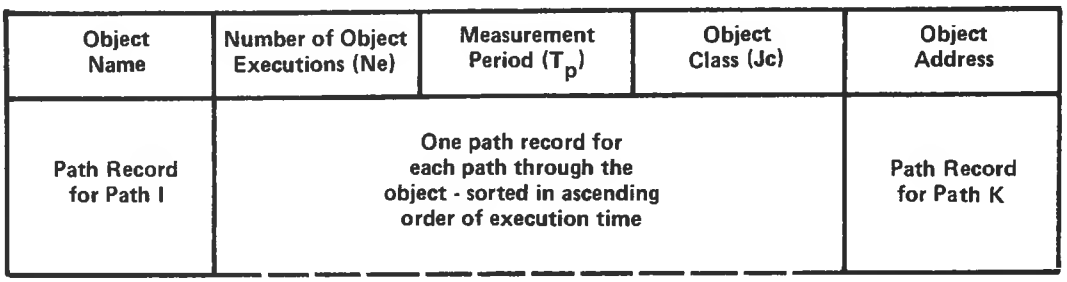


Figure 6.3 Object Record - Data Structure for recording information about an object during a measurement experiment

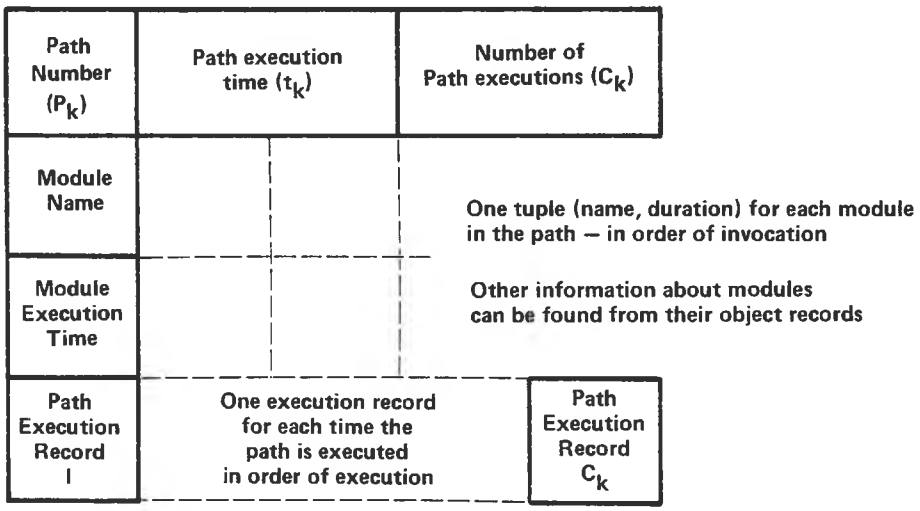


Figure 6.4 Path Record - Data Structure for recording information about individual paths through an object during a measurement experiment

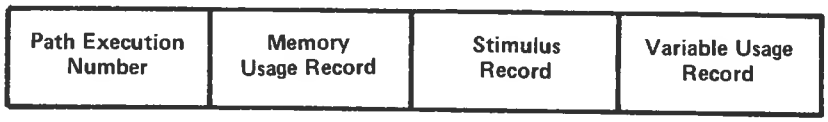


Figure 6.5 Path Execution Record - Data Structure for recording information about individual path traversals during a measurement experiment

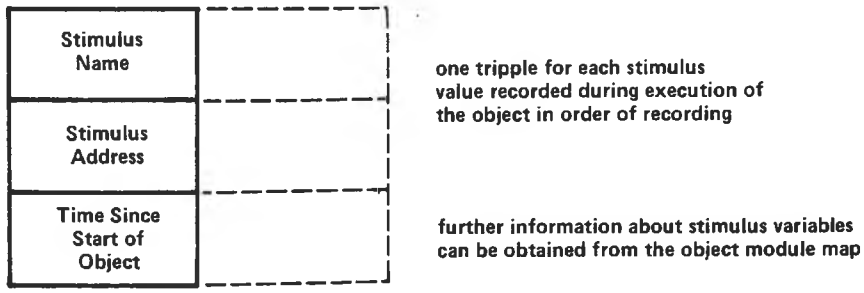


Figure 6.6 Stimulus Record - Data Structure for recording information about individual stimulus variables for an individual path traversal during a measurement experiment

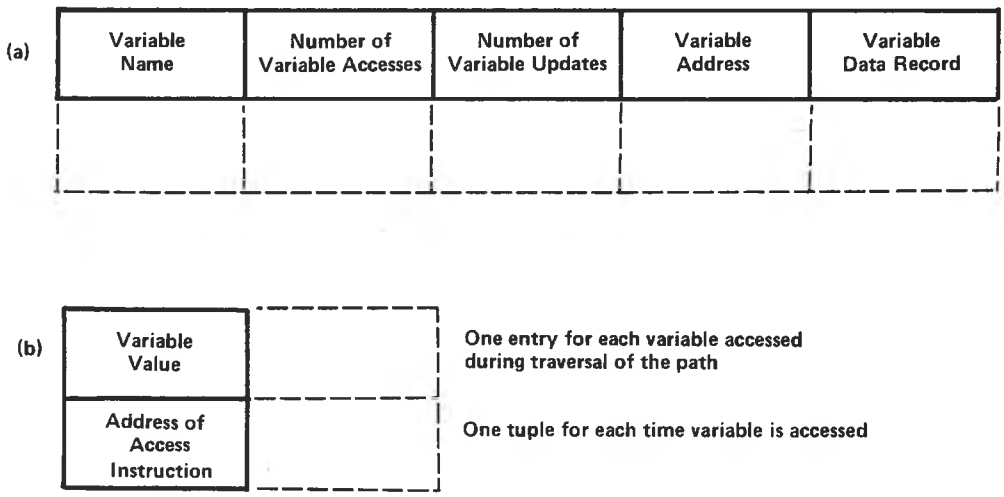


Figure 6.7 (a) Variable-Usage Record and (b) Variable-Data Record - Data Structures for recording operations on a variable during an individual path traversal

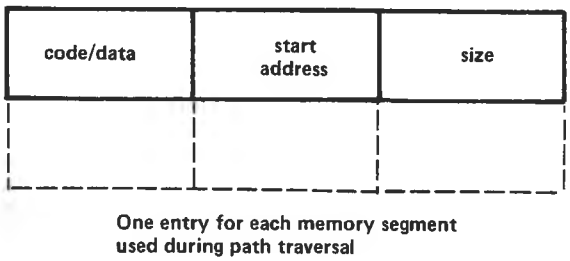


Figure 6.8 Memory-Usage Record - Data Structure for recording usage of memory segments during an individual path traversal

descriptor (for example: old module identifier, pause; new module identifier, start; time relative to start of measurement period). In either case, the analyst must have a reasonable understanding of the system to be able to do the reconstruction. This understanding can be built into the probes, or into the analyser. The latter has two obvious advantages: less interference, and event probes can be added before the system is understood. Understanding can be built up through analysis of measurements. During the analysis of experiments, the analyst has two options: either she can manually reconstruct the information, interactively, from her own knowledge, or she can build the information into a data base in the analyser. Ultimately, by combining this data base and the measurement data base, an expert system could be constructed.

6.1.2. Measurement Algorithms

Most of the measured values (figure 2.2) can be obtained from the event trace (figure 6.2). Other measured values are obtained from stimulus information. The measurement algorithms massage the data in the event trace to produce an **object record** for the object under study. An object record (figure 6.3 to figure 6.8) is a data structure in which all the information measured about an object is stored.

An object record (figure 6.3) consists of an object descriptor, and a set of path records. A path record (figure 6.4) consists of a path descriptor (a sequence of modules, and their execution times) and a set of path-execution records.

Every time a path is executed, a path-execution record (figure 6.5) is generated. The path-execution record includes a stimulus record (figure 6.6), a variable-usage record, a variable-data record (figure 7.7), and a memory-usage record (figure 6.8) for that invocation of the path. The set of path-execution records can become very large, very quickly. Thus, if the experiment doesn't require data about individual path executions than the path-execution record should be left out of the path record. An object record that doesn't include the set of path-execution records grows to a fixed size, and thus, can be used to collect data over an extended period of time (until the values assigned to count variables overflow).

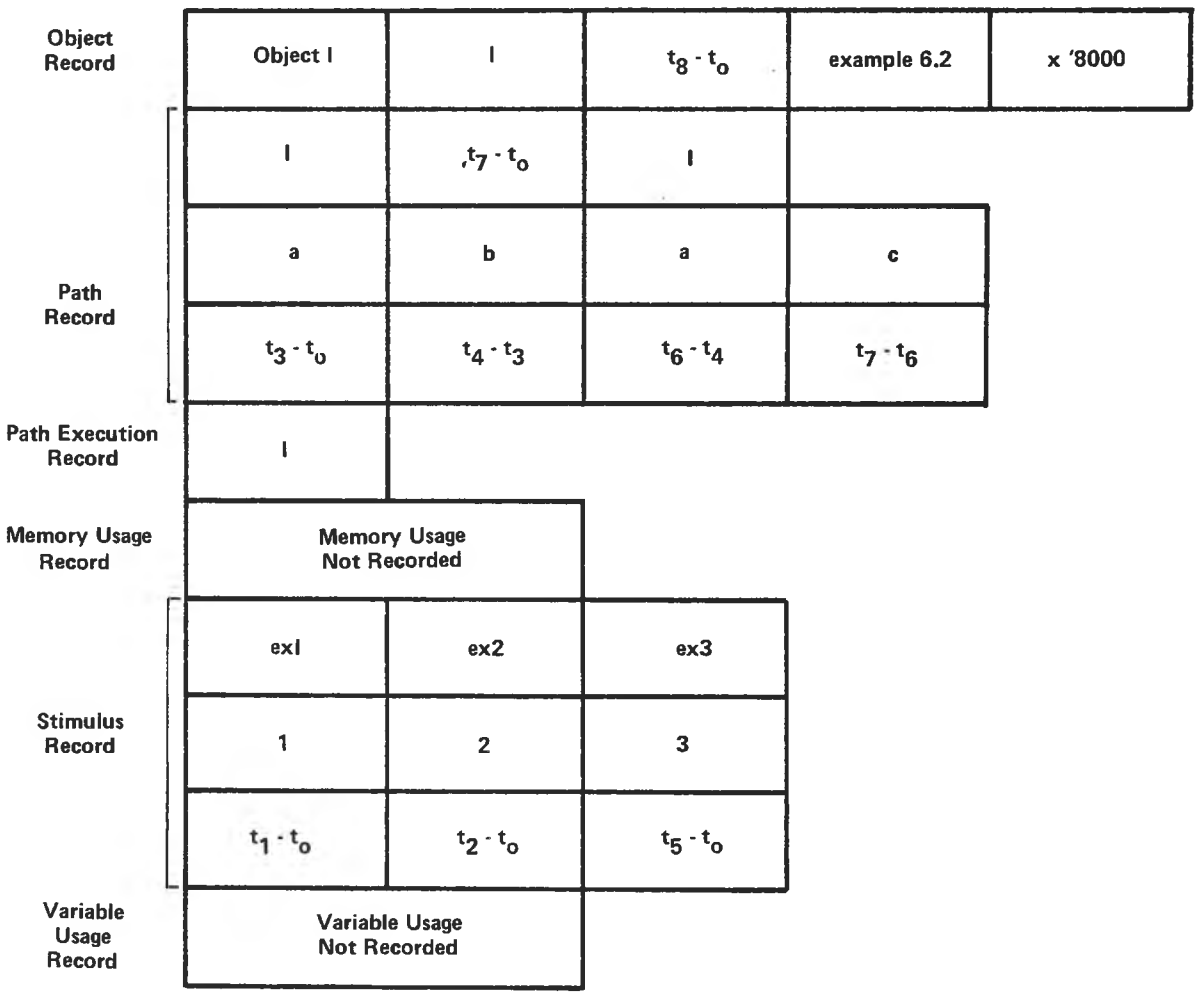


Figure 6.9 Object Record for the event trace in figure 6.2

The data structures, shown here in diagrammatic form, can be implemented in a number of ways, although the representation implies a linked list of variable sized structures. This set of structures enables the recording of all data, and some measured values derived from the data, without data loss. As a consequence, they can become large and cumbersome. To minimise this problem, only those data structures needed for an experiment should be used, and data reduction should be done in situations where data loss is not a problem. I have not implemented a full set of data structures in any experiment, and in some experiments I have generated and recorded the values manually. In the construction of a hybrid monitoring-tool, based around a logic-state analyser (section 6.6), a subset of the object record has been implemented (figures 6.13,6.14).

The object record for the event trace in figure 6.2 is shown in figure 6.9. Calculated values (section 6.1.3) are derived from the information stored in the object record. An additional data display which could be added to the set of displays, described in section 2.8, is a display of an object record, or parts thereof (figure 6.14). Methods for setting up the memory usage record, and the variable (data structure) access record are discussed in a later section, because they may require extensions to the event trace.

Measured values can be obtained from the event trace, and stored in the object record, using the following algorithms:

A6.1 The measurement period (T) is the difference between the times recorded in the first and last tuple in the event trace (e.g. $t_8 - t_o$ in figure 6.2)

A6.2 An execution path (p_k) through an object is the sequence of modules between initiation and termination of the object (e.g. a, b, a, c for object 1 in figure 6.2)

A6.3 The execution time (t_k) for that execution path is the difference between the times recorded in the start and finish tuples of the object (e.g. $t_7 - t_o$ in figure 6.2).

The names of objects, modules, and variables are obtained from either the analyst, interactively, or from the data base of information about the target system (including the module map) that

has been constructed by the analyst in the monitoring system.

The **Object-Record-Generation algorithm** obtains the set of execution paths (P_e), the set of execution counts (N_c), the set of execution times (T_e), the module sequence, and the stimulus information from one pass through the event trace. Collected data is stored in the stimulus record, the path-execution record, the path record, and the object record. This algorithm can be used to analyse the event trace in real-time, as event tuples and stimulus triples are generated.

```
A6.4  Go to the start of the trace
      Initialise object record
      Measurement period start time = time in first tuple
      While there are still invocations of the object to be counted do
        initialise a temporary path record
        move to the next object traversal to be analysed
        object start tuple = first tuple
        module start tuple = first tuple
        while object traversal not complete do
          read next record on event trace
          if stimulus information then
              calculate time since start of object
              save a triple in stimulus record
          else {event information}
              if end of an object traversal then
                 calculate module execution time
                 save module tuple
                 calculate object execution time
                 if a new path then
                     add a new path record to object record
                     copy temporary path record to new path record
                     set path counter to one
                 else copy appropriate data from temporary path record
                         to actual path record
                     increment path counter
              else {end of a module}
                  {expand here to detect interruption by foreign modules}
                  calculate module execution time
                  save module tuple
                  update module start tuple
      {end of measurement trace}
      Sort path records in ascending order of execution time
      Calculate measurement period
      Calculate total number of path executions
```

Algorithm 6.4 does not detect the pause and resumption of modules due to interruption. This can be done by post processing, to combine module fragments, or by expanding the algorithm

to include a **foreign module handler** (algorithm 6.5). The choice of which method to use in a particular experiment will depend upon the likelihood of interruption, and the analyst's knowledge of the system.

In some experiments the analyst may wish to study the activity during pauses. This can be done either by recording the module sequence during the pause, or if the cause of the interruption is known, by studying the interruption as an object.

Algorithm 6.5 detects interruptions by detecting modules that should not normally occur in the module sequence. When first measuring a system, the analyst may not know enough to specify the expected modules in the sequence, although this information should be obtainable from the object's module map. A switch is included to allow the analyst to select foreign module removal as required.

A6.5
```
if foreign-module-removal switch set then
    if next module is a foreign module then
        if foreign-module-detected flag not set then
            set foreign-module-detected flag
            save tuple time as foreign start time
    else {not a foreign module}
        if foreign-module-detected flag set then
            {end of foreign module sequence}
            clear foreign-module-detected flag
            calculate execution time of foreign modules
            add execution time to time in module start tuple
            add execution time to time in object start tuple
        else {end of a normal module}
```

The set of execution times for each module (T_m), the set of execution paths for each module (P_m), the set of execution counts for each module (N_a), and the total number of module executions (N_m) can be obtained by applying algorithm 6.4 at lower levels in the hierarchy. Remember, the term module was only introduced as a shorthand term for object-at-the-next-lower-level.

Counters should also be available for counting the occurrences of specific values of stimulus information etc. Use of stimulus information, memory usage, and quantity of data is

looked at in the following section.

Object paths are differentiated by path execution time, module sequence and execution times of the modules in the sequence. At times only one of these may be used to distinguish paths, or execution time ranges may be used to reduce the number of distinct paths by considering similar paths as one path. It is possible for different paths to have the same execution time.

It is also possible for one path to have different execution times. This creates a problem for the analyst, however, investigating the cause of varying execution times, for the same path will lead to increased understanding of what is actually happening inside the computer. Execution time variations can be due to a number of different causes:

- The execution time of some assembler instructions, particularly on older machines, is data dependent (see section 5.4 for an example).
- Temperature variations in either the processor or the monitoring tool can cause apparent variations in execution time.
- In systems with asynchronous memory busses, memory access time may vary between different memory banks due to the usage of different types of memory. Thus, the place the program is loaded into memory may cause execution time variations.
- If a direct-memory-access controller steals memory cycles during the execution of the object, the execution time will increase. This can be compensated for either by disabling DMA during the experiment or by counting DMA cycles, and associated dead cycles.
- The device which will cause the greatest headaches is cache memory. The execution time will depend upon the cache hit ratio, which may vary from execution-to-execution depending upon process switches, and interrupt handlers, pausing and resuming object execution. The simplest way to overcome cache problems during experiments is to switch it off. However, such measurements may not be what is required by the performance study, although, they can provide a yard stick for studying performance improvement through

using cache. The only real solution is to count both the number of instruction cycles and the number of cache hits or misses during the execution of the object, and use these to normalise the execution times. In many studies, cache variations may not be worth worrying about.

To overcome the effect of minor variations in path execution times when deciding if the current path is a new one or not, a range of execution times can be used. This should only be done after the cause of the variation has been established.

Object class is obtained either as a piece of stimulus information or from a computer (or human) data base of object information.

6.1.3. Calculation Algorithms

From the information stored in the object record, a number of values are calculated. Calculated values can be obtained using the following algorithms:

A6.6 The set of object-execution frequencies (F_e) is obtained by dividing the set of path execution-counts by the total number of object executions. The set of module-execution frequencies (F_m) is obtained in a similar manner.

A6.7 The object throughput (N_o) is obtained by dividing the number of object executions by the measurement period. Module throughput is calculated in the same way.

A6.8 The set of normalised relative-path-execution times (T_r) is obtained by dividing the execution-time for each path by the maximum execution-time. The execution-time of the last entry in the object record is the maximum execution-time, because the path records have been sorted in ascending order of execution-time (algorithm 6.4). Due to the recursive nature of the object definition, the set of normalised relative-module-execution times can be found using the same algorithm.

A6.9 The set of module-to-path execution time ratios ($E_{\nearrow}$) is calculated by dividing the execution time for each module in the path by the execution time of the path.

A6.10 The set of path utilizations (U_p) is calculated by multiplying the path execution time by the number of path executions and dividing the result by the measurement period. Module utilizations are calculated in a similar manner (equation 2.42).

6.2. Data Display

In section 2.8, a number of data displays were discussed. All of these are graphical representations of combinations of the measured and calculated values. Thus, they display the contents of the object record, and values derived from the contents of the object record, in a form that is easily analysed by people.

A6.11 To plot an object spectrum:

```
Determine vertical axis scaling from the range of execution counts.
Determine horizontal axis scaling from the range of execution times.
For each execution path
    Draw a vertical line, with length equal to the execution count at
        the horizontal position equal to the execution time
        - this is a spectral line.
    Write the execution time on the spectral line.
    Write the stimulus information on the spectral line.
Add titles etc. to graph.
```

As with the production of the object record, data display can be done by programs running on the monitoring tool or by manual methods. The choice will depend upon the tools available, the purpose of the experiment, and the amount of data to be analysed.

6.3. Memory Usage and Variable Access Measurement

Measurements that relate the object to physical-memory addresses require an extension of the event trace to include memory-bus activity. Variable accesses can be determined with software probes, but hardware probes create less interference. In order to use hardware probes, the physical address of the variables must be known. Variable accesses must be counted and independent counts maintained for reads and writes to individual variables or data structures.

Thus, while recording the event trace, the tool must be able to detect accesses to variables, or to specific memory addresses. Measurement of memory usage also requires the monitoring of accesses to memory. Memory usage can be recorded with a hardware tool, by monitoring the address-bus activity during execution of the object. On many computers, control signals can be used to separate data accesses and instruction fetches.

Recording and analysing all address-bus activity to determine which memory segments are used gives a very precise measurement, but requires a lot of resources. Often this level of precision is not needed, and coarser measures can be used. One simplification is to segment the memory into fixed-size blocks, and simply record access to any location in a block.

At higher levels in the object hierarchy, memory usage can be measured either from the event trace or with stimulus information. At the block-structure level the routines which access dynamic data-structures (e.g. add to list) can be defined as modules. By counting and comparing module executions ,with respect to time, the number of data elements in a data structure at any time can be calculated. Alternatively, the size of the data structure could be recorded as a piece of stimulus information every time the structure is updated.

At the task level, two of the modules that the system can be decomposed into are: the page-fault module, and the memory-allocation module. To study the usage of various memory segments in the working set, a segment identifier is output as stimulus information by the memory-allocation module. By counting the executions of these modules during the execution of an object (and when setting up to execute the object), a trace of an object's working-set with respect to time can be produced. Alternatively, the number of pages in an object's working-set could be recorded as a piece of stimulus information, every time it is changed. Algorithms for the measurement of working-set behavior are discussed by Sprin (1977) in chapter 5 of his book on program behavior.

Recording the address at which a probe is inserted in an object, together with the event tuple, can also give a measure of memory usage. One problem to avoid is that you may record

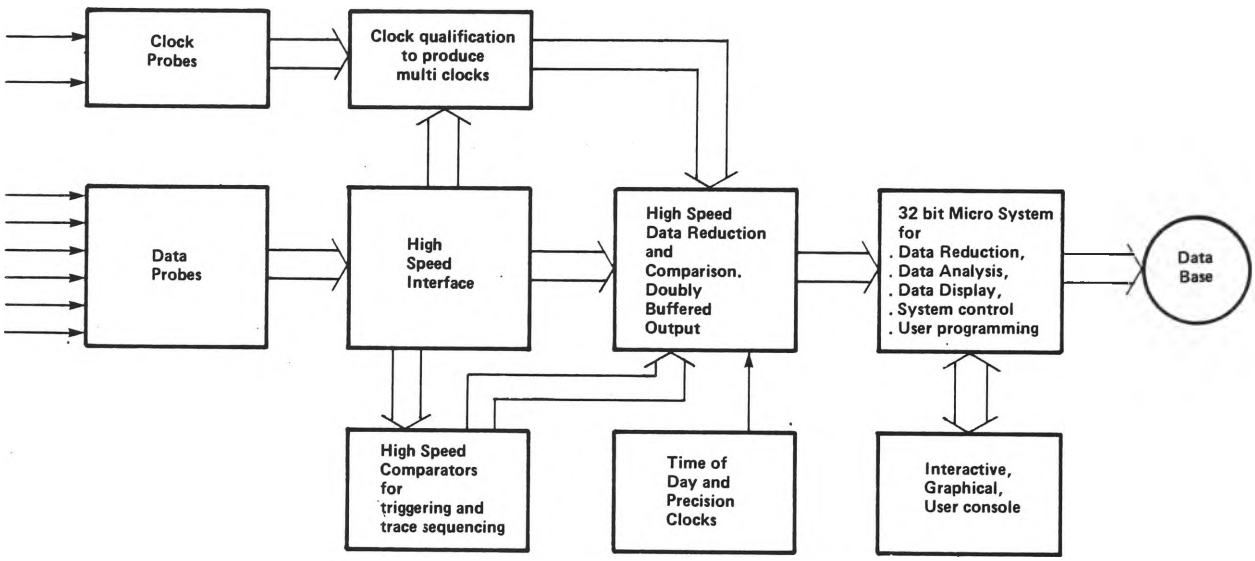


Figure 6.10 Hybrid Real-Time Performance Analyser

the probe address instead of an address in the module (section 7.3.3).

These measurements are one of the possible uses of general-purpose probes, and counters, not permanently allocated to the event trace. In a hardware tool, this means extra probes to monitor the target computer's bus, and additional event filters. In a hybrid tool, it means the ability to detect and count specific events, or specific stimulus variables.

Measurements of memory usage and variable access are very dependent upon the experiment. One use of these measures is illustrated in the debugging case study in section 7.5.2. Kearns et al (1982) report on the study memory usage, and the performance of memory-allocation procedures, in the context of dynamic data-structures for coroutines. A major use of memory-usage measures is in the performance evaluation of virtual-memory systems (Ferrari 1976); particularly in the areas of program referencing-behavior, and page-fault replacement algorithms.

6.4. Hybrid Tool Design

In the rest of this chapter, the design of a hybrid tool (figure 6.10) will be discussed. In the previous sections, the functions performed in the data reduction, data analysis, and data display sections of the tool have been described in detail. Questions relating to class of tool, type of probes, insertion of probes, event detection, synchronisation of monitor to target process, and realisation of a monitoring tool will be discussed in the framework provided by the measurement formulation. Also, an actual tool, as opposed to the ideal, is described.

A hybrid tool has been chosen for a number of reasons. Hybrid tools have demonstrated their superiority over purely hardware or purely software tools (section 4.5). Although many of the measures indicated by the performance measurement formulation can be made with independent tools, a hybrid tool is required to implement the full set. Also, when discussing a hybrid tool, both hardware and software tools are discussed.

Hybrid tools minimise the amount of modifications that have to be made to the target system. The software added to the system is reduced to probes, and the generation of an object

data base. By careful design, modifications to insert hardware probes can be minimised. Consequently, the cost of the required modifications is reduced, and the bulk of the measurement tool can be used on other systems, amortising the tool cost over a large number of measurement experiments. All new computers are being designed with LSI and VLSI chips. As a result, many of the signals that were available to hardware tools are no longer available. Software tools have to be used to compensate for the reduction in hardware information.

Research projects are often directed by circumstances. Prior to the research detailed in this dissertation, a logic-state analyser had been bought for performance evaluation.* Thus, an obvious research direction involved using it. The availability of sophisticated, relatively low-cost, logic-state analysers will dictate their increased use as performance-evaluation tools. As this research is about formulating a unified body of performance-measurement knowledge (theoretical and practical), tools discussed in this context should be constructed from readily available components. Otherwise, the application of the research will be restricted to the few who can afford specialised tools. Also, by using off-the-shelf components, the cost can be kept down, and obsolescence can be overcome simply by stepping up to the new generation of tools. Finally, such research should impact the design of new generation tools, and hopefully, improve them.

6.4.1. Philosophy of Hybridisation

A workable philosophy of hybridisation should take into account both implications of the performance measurement formulation and practical considerations based upon the complementary nature of hardware and software tools. Obviously, we need to use software tools to get software specific information, and hardware tools to get hardware specific information. Also, hardware tools must be involved when measuring parallel information. In each situation probe

---

* The purchase of the logic-state analyser was funded by the Department of Science and Technology, Australian Research Grants Committee.

selection is guided by: availability of signals, minimum interference, required precision, and sufficient resolution.

From the measurement formulation, it is the probes which produce the event trace and stimulus information. The time stamp is added by the data reduction section. At different levels in the hierarchy, different probe realisations and placements are needed. Also, a simple way of determining which level in the hierarchy an event in the trace represents is required. Part of the problem here is the requirement of high precision (hundreds of nanoseconds) over a wide resolution range, from microscopic modules (individual instructions) to macroscopic objects (complete systems).

The rate at which events occur decreases as you move further up the hierarchy. Every time an event record, or a stimulus record, is added to the event stream, a clock signal is generated to indicate the presence of new information at the input of the data-reduction section of the tool. An obvious way of indicating the level-of-the-record in the hierarchy is to use separate event clocks for each level. The process of data compression in the event stream, section 6.1.1, requires that when several events occur at the same quantum in time the event clocks are synchronised, and the event record contains information pertaining to all the events. Time quantums are defined in terms of the finest clock, usually the machine-code-execution-cycle clock.

At the **machine-code level**, the event stream can be measured with a hardware probe. The event clock is the instruction-fetch clock, and the stimulus clock is the data-fetch clock. Hardware probes are attached to the target computer's address, data and control busses. On the active edge of the event clock, the address and operand code of the current instruction (instruction being fetched) are on the computer's bus. On the active edge of the stimulus clock, the address and contents of a variable are on the computer's bus. The problem of generating separate event and stimulus clocks on a microprocessor is discussed in section 8.3. Measurements at this level are usually made in conjunction with a program listing and a program load map. If these are not available, the techniques used at higher levels in the hierarchy must be used.

At the **high-level-language level**, the above approach is very difficult to use, because it requires a knowledge of the code generated for each high level instruction. Some compilers (section 5.5.1) produce an assembler listing at an intermediate stage in the compilation, which can be used for machine-code-level type measurements. At this point, software probes become easier to implement than hardware probes.

However, there are a number of ways of handling this problem, based upon the way in which you choose to hybridize. The compiler can be modified to produce sufficient information for a hardware tool to make the measurements. This method has the advantage of no interference during the measurement, but requires a modified compiler before measurements can be made, considerably restricting the generality of the approach. As our ultimate goal is the integration of general-purpose monitoring tools at design time, or at least the design of a cheap tool that can easily be added to an existing system, we seek to use as little of the target system's resources as possible.

The approach to hybridisation which most closely meets our design goals is the use of software probes to pass information to the hardware tool. An instrumented system is permanently fitted with low-interference software probes. Under normal system operation, system probes automatically provide information about user programs for task level measurement and accounting. This can be done without inserting probes into the user process.

High-level language programs can be compiled and then executed, compiled into pcode and the pcode interpreted, or interpreted statement by statement. Probe insertion depends upon the method of program execution. When a program is to be compiled, probes have to be inserted between each high-level statement to achieve complete decomposition. This is tedious and creates considerable interference, hence, it is only done for special experiments (for example measuring high-level-language-instruction usage and debugging). An interpreter will execute at least one common piece of code every time an instruction is executed. Placing a probe in this piece of code will instrument the program without modifying the program. An obvious choice for module identifier is the line number of the high-level statement. A code representing the

type of high-level instruction could be written as stimulus information.

For compiled code, measurement at the **block-structure level** makes more sense. Sequential pieces of code are considered as a block, and probes are inserted at the start of each block. Stimulus information is usually related to the function of the program. Measurement at this level is discussed in the next chapter. Once the extent of the object has been measured, and related to the object's function, probes at this level should be removed as they create considerable interference. Techniques at this level can be applied to machine-code programs as well as to high-level language programs.

At the **program-execution level**, individual programs, processes, and routines are instrumented. Probes are inserted to indicate the start and end of each routine. System programs are often instrumented permanently at this level. The operation of a system can be observed as a sequence of events at this level. The level is low enough to give a good clear view of system operation and high enough not to get bogged down in detail. An assumption made here is that routines are structured properly with one entry and one exit point. A 'spaghetti' coded operating system can be a measurement nightmare.

Stimulus information plays a very important role at this level. Many routines within an operating system can be called by any one of the currently active processes. Stimulus information can be used to determine which active process called a routine. For example, the stimulus information for a reentrant terminal driver would be the terminal number, and the characters handled by the routine. The stimulus information for a run-time-library routine would be the identifier of the calling process; and the stimulus information for a disc-file handler includes the file name, calling process identifier, and the track number. In the latter case, stimulus information gives meaning to the operation being performed by the handler.

At the **task level**, we wish to study the system as it executes a user or system task. The instrumentation at the program-execution level gives us the sequence of modules in the task object. What we want to be able to identify now is the start and termination of tasks. During

the life of a task it can exist in a number of states. The program which controls the allocation of CPU resources to a task is the scheduler. Thus, the appropriate place to insert a probe is in the scheduler (for example the OASIS task monitor - section 4.5.1). The module identifier in the event tuple is the process number allocated to the task by the system. Task state and job class are stimulus information. Other resource control programs, for example the control of memory page allocations, can also be instrumented.

This instrumentation enables the study of individual tasks, tasks of specific job classes, and the measurement of system resource usage by tasks. Thus, it could be used for accounting purposes. A task can be viewed as a collection of modules which interact to perform a desired operation. Measurement at this level can give the analyst insight into the complex web of interactions which occur inside an operating system.

At the **system level**, the lower-level instrumentation can be used to study overall system operation. The system object decomposes into a collection of tasks. Resource usage by job class, impact of multiprogramming level, and detection of bottle necks are typical of analysis done at this level.

At the system level we once again introduce hardware, as opposed to hybrid, probes. Signals such as CPU busy and CPU mode (kernel or user etc.) can be monitored with hardware probes to measure CPU utilization and the system overhead. Software probes can also be used for CPU busy type measures. For example, in an interrupt-driven operating system, probes could be placed in the idle loop and in the interrupt handlers. Another use for software probes is the measurement of resource queues as stimulus information.

6.5. Desirable Features of a Hybrid Tool

An ideal tool is one with infinite precision, infinite resolution, zero interference, a friendly user interface, and enough power to handle any measurement situation with ease. Unfortunately, the gap between the ideal and the actual is so large that a study of the ideal gives us little help in improving actual tools. Studying the ideal does fix desirable goals in our mind and gives our

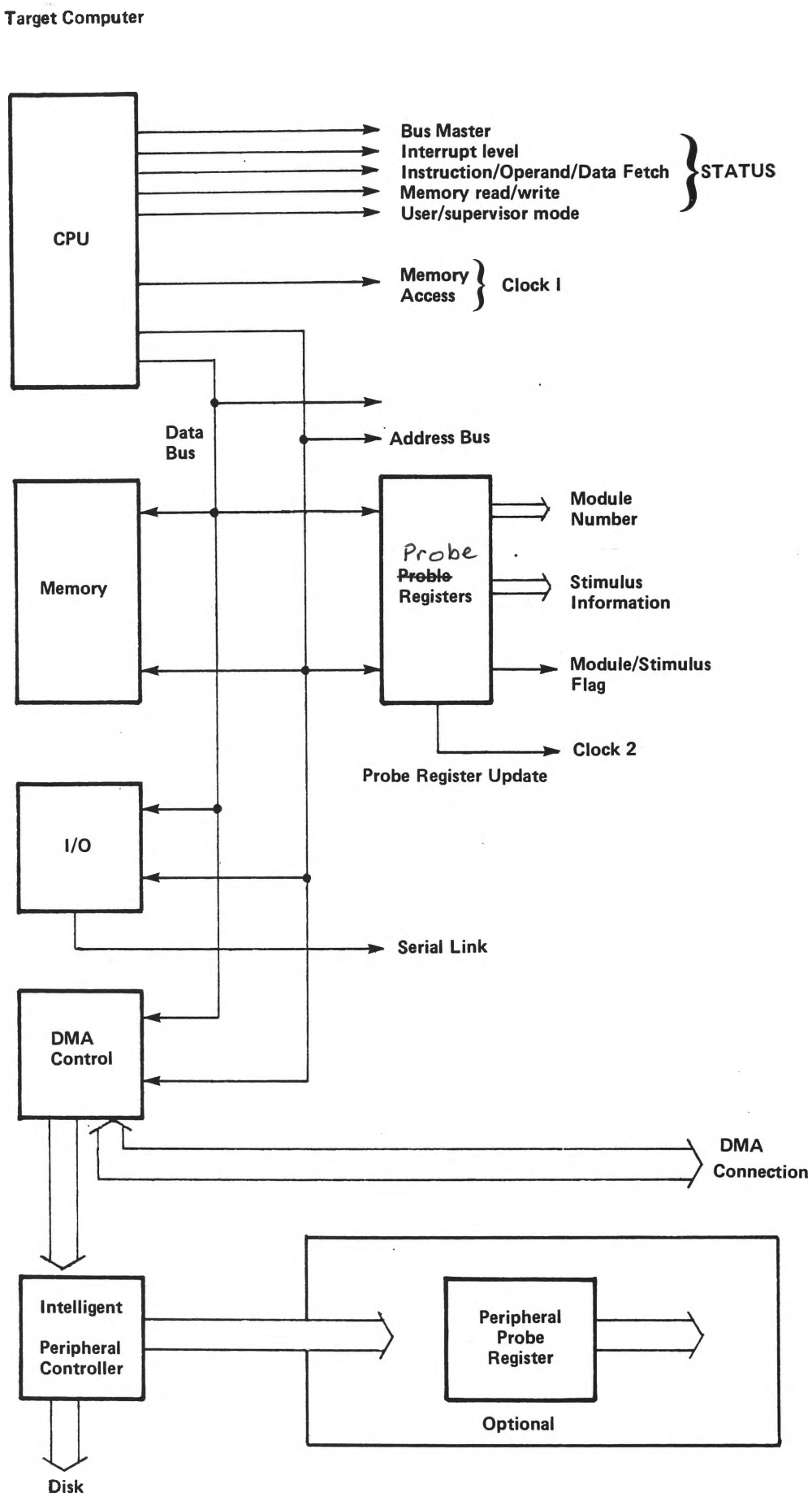


Figure 6.11 Hardware Probes

work direction. Having briefly considered the ideal, we will now look at desirable tool features that can be achieved with the current generation of technology.

6.5.1. Hardware Section

The **hardware probe** to be added to the system under measurement is a simple digital-output-port (figure 6.11) and some connectors. The output port consists of a sixteen-bit module-number monitor-register and a thirty-two-bit stimulus monitor-register; each with a light emitting diode display and a connector. The stimulus monitor-register can be used to store a single thirty-two-bit variable, two sixteen-bit variables or four eight-bit variables. The light emitting diodes are for visual analysis and user confidence only.

Software can store information in these registers pertaining to program operation at any time. When either probe-register is updated a clock pulse is sent to the analyser. Software overhead is less in a hardware system that uses memory-mapped input-output than in a system that has special input-output instructions, especially if they can only be executed in supervisor mode.

Other hardware includes connectors for a serial communications port, a direct-memory-access channel (only necessary in some situations) and state-analyser signals, including the address bus and processor status lines: user/supervisor mode, interrupt level, type of memory access, etc.

Two clock signals are included: a memory-access clock, and a probe-update clock. The probe-update clock can be synchronised to the memory-access clock. The active edge of the probe-update clock occurs at a point during the next instruction fetch when the target computer's busses are stable. Thus, a hardware tool reading the probe in response to the probe-update clock can also read the address of the next instruction.

The hardware tool is able to handle multiple clocks in parallel. These clocks can all be generated externally, or preferably, they can be generated from the two clocks using appropriate signals as clock qualifiers. The memory-access clock can be decomposed into a number of

clocks representing different bus activity if signals which describe bus activity are available. Typical bus activity includes: instruction fetch, operand fetch, data fetch, data write, I/O operation, interrupt vector, dead cycle, and dma activity.

The probe-update clock can be decomposed into event and stimulus clocks, either with a flag generated by the probe or by comparing successive events to see which monitor register changed: module identifier or stimulus data. The former is simpler to implement. The probe-update clock can also be decomposed into separate clocks for each level of the hierarchy (block-structure, program, and task) by using high order bits of the module identifier as level descriptors.

Thus, the clock qualification section of the hardware tool should be able to read two synchronised clocks and decompose them into a number of slower synchronous clocks. In addition, it should be possible to combine any of these clocks using standard logic functions, or counters, to produce the desired measurement clocks.

Once the clocks are generated, they are used to synchronise the data capture circuits with their associated data values in the continuous event stream presented to the hardware probe. Captured data is analysed in the data reduction circuits to detect events of interest to the current measurement experiment. Data recording triggers are generated by detecting sequences of events. Sequence detection may involve counting the number of occurrences of some of the events in the sequence. Sequence detectors and counters should be available for each clock in use, so that a number of parallel triggers can be generated. These triggers can be used separately, or combined using logic functions, to initiate data recording.

Data recording is controlled by comparators and counters associated with a particular clock. When an event of interest is detected, it is recorded. The clock used for data recording does not have to be the same as the clock used for triggering recording. Thus, events at one level can be recorded after the occurrence of events at another level. Parallel data-recording-filters enable the recording of events from various levels, and of different types, in the one

event stream. Obviously, the hybrid monitor must include user friendly tools for easy definition of events-of-interest and clock qualification.

Another desirable feature of an event filter is the ability to record events near the event of interest. For example: record six events before the event of interest, or record the seventh event after the event of interest. I do not know of a logic-state analyser that has this level of event filtering and clock qualification. Some of the newer models have a few of these features. However, this is the direction the design of future logic-state analysers should take.

With advances in technology, the depth of the data storage buffers in logic-state analysers has increased considerably. Early models had limited storage (typical 64 words), and limited width (16-32 bits). Data was stored in the buffers until they were full, and then transferred to a separate analysis computer. During the transfer, incoming data was lost. By analysing the data as it is recorded, and with reasonably deep or double buffers, continuous recording over long periods of time should be possible. This requires a high degree of parallelism in the analyser and fast transfer to backup storage.

The difficulty of inserting the hardware probe varies from system to system. Usually the probe can be built on a card that plugs into the target computer's backplane. Using a probe card, designed for the system, minimises errors due to faulty connections, eliminates probe placement errors, and considerably simplifies instrumentation. Some systems do not have all the desired signals available on the backplane, and may require modification. For one off measurements, a probe register can be simulated by writting to a memory location.

In systems that don't have a backplane, the probe has to be built into a separate module which is connected to the processor with a logic clip. This module is an extension of the logic-state-analyser-personality-module concept. One approach to building a general-purpose, system-independent, microprocessor-specific hardware-probe is to add monitor registers to the standard personality module. Thus, the probe would consist of a personality module, connected to the target processor with a logic clip, which contains processor-addressable monitor-registers;

a state machine to produce bus-status information for clock qualification; and connections to all bus signals. A method of selecting suitable addresses, in the input-output address-space of the target system, for the monitor registers is needed. The probe electronics could be powered from the target processor.

Memory management and cache memory create significant problems for probe connection. The probe must be connected to the address bus after relocation from virtual to physical address space and before cache memory, otherwise the addressing information will not reflect program execution. Also the memory-read status signals must represent processor requests not cache requests. On some computers, the cache memory is an integral part of the processor and the address bus is only accessible after cache. Thus, the address bus does not contain the address of the current instruction during a cache hit. On a cache miss several instructions (which may not be executed) after the current one are read into cache, and again the addressing signals to main memory may not reflect program execution. Microprocessors which include integral cache (for example the proposed m68020 and z80000) are going to be very difficult to measure at the machine-code level.

Adding an additional status signal which indicates whether the current memory reference is a cache hit or miss allows cache performance for instruction fetches, operand fetches and data fetches to be measured for individual modules. Once again, the need to define performance measurement requirements at design time and build them into an integrated hardware/software design is emphasized.

6.5.2. Software Section

Software probes are used to write module identifiers (event descriptors) to the module-number monitor-register, and stimulus information to the stimulus monitor-register. A method of probe insertion is required to enable users to instrument their programs, and for the initial instrumentation of system programs. Probes can be inserted into program source manually, automatically, or with an interactive tool. Manual methods can be laborious, but for some experiments

the expense of porting a probe-inserting program to the target system may not be justified. Automatic probe insertion is possible with sophisticated tools, but the analyst has little control over where they go, and the use of automatic tools may lead to less understanding rather than more.

An interactive tool has the advantages of automation while allowing the analyst complete control over probe insertion. The tool should enable the analyst to search the program text sequentially and insert probes as desired. As probes are inserted, the module identifier should be allocated automatically and an object module map constructed. Probe software can be a standard piece of code, or within certain constraints a user defined routine. Stimulus probes have to be inserted interactively, or manually, because the analyst has to select the stimulus variables. In some of the reported integrated instrumentation environments (chapters 8 and 9) tools of this nature have been included.

Thus, one of the tools included in an instrument is a portable probe-editor. The system dependent parts of the probe-editor (for example address of hardware probe) must be easy to set up. This editor should maintain the module map, as part of the object data base, for the target system. A second tool is required to enable the analyst to study the object data base.

In single-task operating-system environments, all activity during the execution of an object has to do with that object. Simply inserting probes at the start of each module is enough to give accurate measurements. In some closely controlled experiments on multi-tasking systems, instrumentation in this way is adequate also. However, in any system where an object can be interrupted, a mechanism is required to write the module identifier to the probe register when the interrupted module resumes. Note, as the module-number monitor-register is only updated when a module starts or resumes, the module identifier is accessible to the external tool while ever the module is executing.

To handle the pause and resumption of modules, it is necessary to save the module identifier of the interrupted module so that it can be written to the probe on resumption of the

interrupted module. If a system is understood well, it is possible to do this by inserting additional probes at the end of modules that can interrupt other modules. One difficulty here is the possibility of nested interruptions.

A more general way to handle interruption, but unfortunately a way that creates more interference, is to use a module-identifier stack. The module-identifier stack may be a separate stack allocated for that purpose, the stack of the currently executing process, or the stack frame of the currently executing module. The last entry on the stack is always the identifier of the currently executing module. Above the block-structure level there are four ways of entering and exiting a module:

- absolute branch in and absolute branch out,
- jump to subroutine and return from subroutine,
- external interrupt and return from interrupt, and
- software interrupt to a trap handler and return.

In well structured code these pairs occur together, but not always. At levels higher than the block-structure level, the sequence of modules that an object decomposes into can sometimes be seen as a nested set of modules (figure 2.4), if structuring is adhered to. At other times the sequence of modules is discontinuous due to process pre-emption (figure 6.14). These views do not alter the performance measurement formulation, they just provides alternative perspectives on how the modules are ordered.

The algorithms used to handle the module-identifier stack depend upon the way the current module is entered and exited. Thus, the probe routine has to be selected according to the method of call and the method of return. The standard probe algorithms are:

A6.12 On entry to a module by an absolute branch
Pop the module identifier of the terminated module
Push the module identifier of the executing module
Write the module identifier to the probe.

A6.13 On exit from a module by an absolute branch
do nothing

A6.14 On entry to a module by an interrupt
Push the module identifier of the interrupting module
Write the module identifier to the probe

A6.15 On exit from an interrupt sequence
Pop the module identifier of the terminating module
Read the last entry on the stack without changing the stack pointer
Write module identifier (of the resumed module) to the probe

A6.16 When calling and returning from a subroutine, either of the above pairs of probe algorithms can be used, but the members of the pairs cannot be mixed.

If the view is taken that as the subroutine returns to the calling program, the calling module is resumed, then use the algorithms for interruption. If the view is taken that on return from the subroutine a different module is started, then use the algorithms for absolute branches.

The impact of pause and resumption of modules upon the measurement and calculation algorithms has to be considered carefully. If the second view, that of return to a different module, is taken, no problems exist. However, the view that subroutines return to the calling module does create problems. Are the two invocations of the module considered as two separate invocations or as one?

In both cases the path record remains the same, in order to correctly show the module sequence. If the two (or more) sections of the module are considered as one, then the path record has to be massaged by a preprocessor to combine the module fragments before doing the calculations. Taking the alternate position, that they are separate invocations of the same module, means that the algorithms do not need to be changed, but the concept of a module with one entry point and one exit point is violated and completely different pieces of code are considered as one module. All the calculated values, except utilization, will be different with

the two approaches.

The choice of how to instrument subroutines is left to the analyst, however, circumstances may dictate which method is to be used. Treating procedure calls the same as absolute branches has the advantage that the problems of module fragment combination are avoided, but may multiply the number of modules. An event probe has to be inserted in the calling module immediately after the procedure call, rather than in the called module immediately before the return statement. Thus the interference is the same with both methods.

A6.14a On entry to a module by an subroutine call (alternate algorithm)
Pick up the calling-module's identifier from the temporary-storage variable
Push it on the stack
Write the module identifier of the called module to the probe
Set the continuation flag
Save the called-module's module-identifier in the temporary-storage variable

A6.15a On exit from a subroutine (alternate algorithm)
Pop the module identifier of the module being resumed
Save it in the temporary-storage variable
Write it (the module identifier of the resumed module) to the probe

An alternative way of overcoming the problems of treating procedure calls like interrupts is to modify algorithm 6.14 to set a reserved bit in the module number, to indicate continuation (A6.14a), before the module number is written to the probe (figure 6.14). In this way continuation is easily recognisable, and the module can be treated as two separate modules, as in the absolute branch case, or the data can be preprocessed to combine module fragments before calculation. The disadvantage of this method is the increased interference of the probe. Also, in algorithms 6.14a and 6.15a a temporary-storage variable has been introduced, because in some implementations, it is more efficient to use a temporary-storage variable than the top of the stack.

Every system has a **quiescent state**: either the machine goes to idle or it goes to a continuous background process. A special probe is inserted at this point. The purpose of this probe is to clean up the stack, and to assist in the detection of instrumentation errors. If the stack is not

empty, or if it underflows during object execution, then an error condition exists in the balancing of probe pairs. The quiescent state always has a zero module identifier. A module identifier of all ones is the first entry on the stack. If this identifier ever appears at the probe then an underflow error has occurred. By examining the module identifiers immediately prior to the quiescent identifier, unbalanced instrumentation can be detected. Also, this probe is required for system-level measurements of idle time.

A6.17 When going to the quiescent state (separate module-identifier stack)
```
While stack not empty do
        Pop module identifier
        if module identifier  Not  (zero or all ones)
                then write identifier to probe
Push module identifier of all ones
Push module identifier of zero
Write module identifier to probe
```

Algorithms 6.14 and 6.15 (and their alternatives) will work on a single address-space, non-pre-emptive system using one module-identifier stack. However, in **multi address-space systems**, a process can not access memory in another process's address-space, and hence, a separate module-identifier stack is needed for each process. In pre-emptive systems, a separate module-identifier stack is needed for each process to maintain instrumentation correctness.

A system consists of a sequence of tasks, and a task consists of a sequence of programs (processes). Both tasks and processes can have their execution paused by **pre-emptive scheduling**. Typically, an executing process requests service from the kernel through a trap (supervisor call or software interrupt). After the requested service has been performed, the kernel may not return to the calling process; because the process may be waiting for an input-output operation to complete, or a higher-priority process may be waiting to be dispatched.

When pre-emption can occur, algorithms 6.14 and 6.15 will only work if a separate module-identifier stack, or stack frame, is used for each process, just as a separate stack, or stack frame, is used by each process. For example, in the character-handling task shown in figure 2.3 the input-handling process requests the kernel to send a message to the terminal-

administrator process. After the message is sent, the input-handling process is blocked, waiting for a reply, and the terminal-administrator process is dispatched to reply to the message. If only one stack was used, the module identifier ~~poped~~ popped off the stack on exiting the kernel would be the identifier of the paused module in the input-handling process, not the module identifier of the resumed module in the terminal-administrator process (figure 6.14).

An obvious question here is: why is a module identifier ~~poped~~ popped off ~~of~~ the stack, indicating a resumption of the terminal-administrator process, and not a new module identifier pushed onto the stack, indicating the start of the first module in this process. The terminal-administrator process ~~in~~ is in an **infinite loop**, and was blocked waiting for some one to send it a message. User processes are often created to execute once, after which they die; system processes are often created to be eternal, and thus, have to be instrumented accordingly (section 8.4). The entry path to a newly created process is different to the entry path for a resumed process (figure 8.4). In the former case, a new module identifier is pushed onto the stack; in the latter case, a resumed module identifier is ~~poped~~ popped off the stack.

Instrumentation involving several stacks has to be inserted carefully, to maintain the balance between pushes and pops, so that normal program execution is not interfered with. One way of minimising the problem is to use a local variable in the stack frame of each module, and process, as the storage location of the module identifier of the calling process. In this way, the module identifier is always stored in a fixed location relative to the module's return address (figure 8.9), and stack balance is maintained automatically.

When entering the kernel process, the module identifier of the paused module must be saved, either by stacking it or by storing it in the process descriptor of the paused process, so that, when the process is dispatched again the module identifier of the resumed module can be written to the probe. Obviously, on return to the system's quiescent state there is no longer any need to clean up the stack, and a stimulus probe is sufficient.

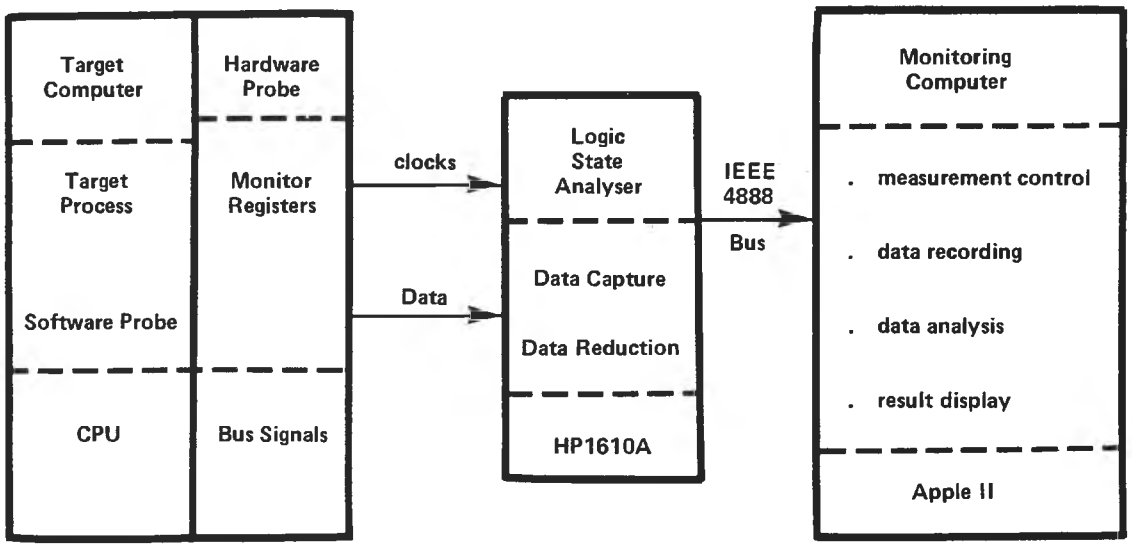


Figure 6.12 Hybrid Monitoring-Tool

A 6.18 Stimulus probe algorithm
Read Stimulus Variable
Write it to probe.

Two other types of programs require special consideration during instrumentation: recursive programs, and re-entrant modules. In both of these cases, the same module can be invoked many times. Probe placement needs to be considered carefully to avoid multiple events with the same module identifier in the recursive case, and to avoid event confusion in the re-entrant case. A number of approaches can be taken, however, the easiest seems to be to use stimulus information. When a recursive program bottoms out, the depth of the recursion can be written as a stimulus variable. When a re-entrant module is entered, the process number of the calling task can be written as a stimulus variable. This stimulus variable can then be used to separate out invocations of the re-entrant module.

6.6. An Actual Tool

During the course of this research, a hybrid tool has been built and used in a variety of experiments. This tool is now obsolete, and only a subset of the desired features was possible, but sufficient has been implemented to validate the performance measurement formulation, and to test the measurement algorithms. The tool is constructed from a logic-state analyser and an Apple II personal computer. Use of the tool as a purely hardware tool was discussed in chapter 5. In chapter 7, monitoring program-execution at the block-structure level is illustrated with a case study which occurred during the development of the hybrid tool. In chapter 8, the instrumentation of a small system is described.

The tool consists of: a probe card (see figures 7.1, 7.2 and 7.3 for an Apple II probe card), a logic-state analyser (figure 5.1), an IEEE 488 bus, and an Apple II personal computer. The interconnection of these components is shown in figure 6.12. As can be seen from figure 7.3, the amount of hardware needed to probe a personal computer is very small. Several software probes are shown in figure 7.4.

```
PROCEDURE GENOREC;
BEGIN (* GENERATE OBJECT RECORD FROM TRACE BUFFER*)
I := 1;
WHILE I<63 DO
  BEGIN (* INITIALISE PATH RECORD*)
  FOR J := 1 TO PLENGTH DO
    BEGIN
    TPATH.MNAME[J] := 0;
    TPATH.MSTIM[J] := 0;
    TPATH.MTIME[J] := 0;
    END;
  WHILE (PROBE[I] <> SMOD) AND (I < 63) DO I:=I+1;
  J := 1; (* FOUND AN OBJECT *)
  MSTIME := I;
  TPATH.MNAME[J] := PROBE[I];
  OBJTRAV := FALSE;
  LMOD := FALSE;
  WHILE (OBJTRAV = FALSE) AND (J < PLENGTH) AND (I < 63) DO
    BEGIN (* TRAVERSE OBJECT ONE MODULE AT A TIME *)
    I := I+1; (*GO TO NEXT TUPLE*)
    IF PROBE[I] = PROBE[I-1] THEN TPATH.MSTIM[J] := STIM[I] (*STIMULUS*)
    ELSE
      BEGIN (* EVENT TUPLE *)
      IF LMOD THEN
        BEGIN (* END OF OBJECT *)
        OBJTRAV := TRUE;
        TPATH.MTIME[J] := TIME[I+1] - TIME[MSTIME];
        TPATH.PETIME := 0; (* PATH EXECUTION TIME *)
        FOR K := 1 TO J DO TPATH.PETIME := TPATH.PETIME + TPATH.MTIME[K];
        FOUND := FALSE;
        K := 1;
        WHILE (OBJECT[K].PETIME <> 0) AND (NOT FOUND) AND (K < OSIZE) DO
          BEGIN (* LOOK FOR A KNOWN PATH *)
          IF TPATH.PETIME = OBJECT[K].PETIME THEN
            BEGIN (*FOUND A KNOWN PATH*)
            OBJECT[K].NOEX := OBJECT[K].NOEX + 1;
            FOUND := TRUE;
            END
          ELSE K := K + 1;
          END;
        IF NOT FOUND THEN
          BEGIN (* NEW PATH, COPY PATH RECORD *)
          OBJECT[K] := TPATH;
          OBJECT[K].NOEX := 1;
          END;
        END
      ELSE (* END OF MODULE*)
        BEGIN
        J := J + 1;
        TPATH.MTIME[J-1] := TIME[I] - TIME[MSTIME];
        MSTIME := I;
        TPATH.MNAME[J] := PROBE[I];
        IF PROBE[I] = EMOD THEN
          BEGIN (*LAST MODULE IN OBJECT*)
          LMOD := TRUE;
          I := I - 1; (*STAY AT THIS TUPLE*)
          END;
        END;
      END;
    END;
  END;
MPERIOD := MPERIOD + TIME[64] - TIME[1];
END;
```

Figure 6.13 Code of a Subset of the Object Generation Algorithm (algorithm 6.4) used in the hybrid monitoring-tool - complete measurement program given in section 13.1

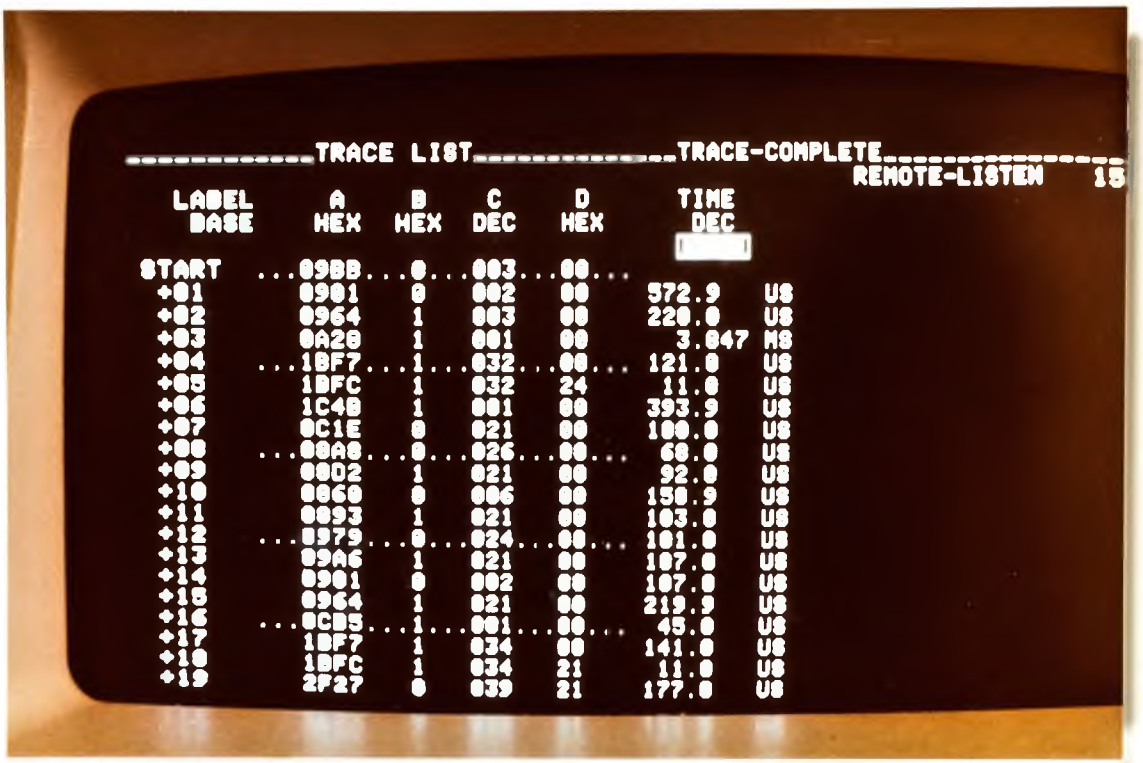


Figure 6.14 Logic-Analyser Display showing the sequence of modules (event trace) during the first four processes of the Character-Handling Task (figure 2.3) prior to modifications. A - address bus, B - continuation flag, C - module number, D - process number.

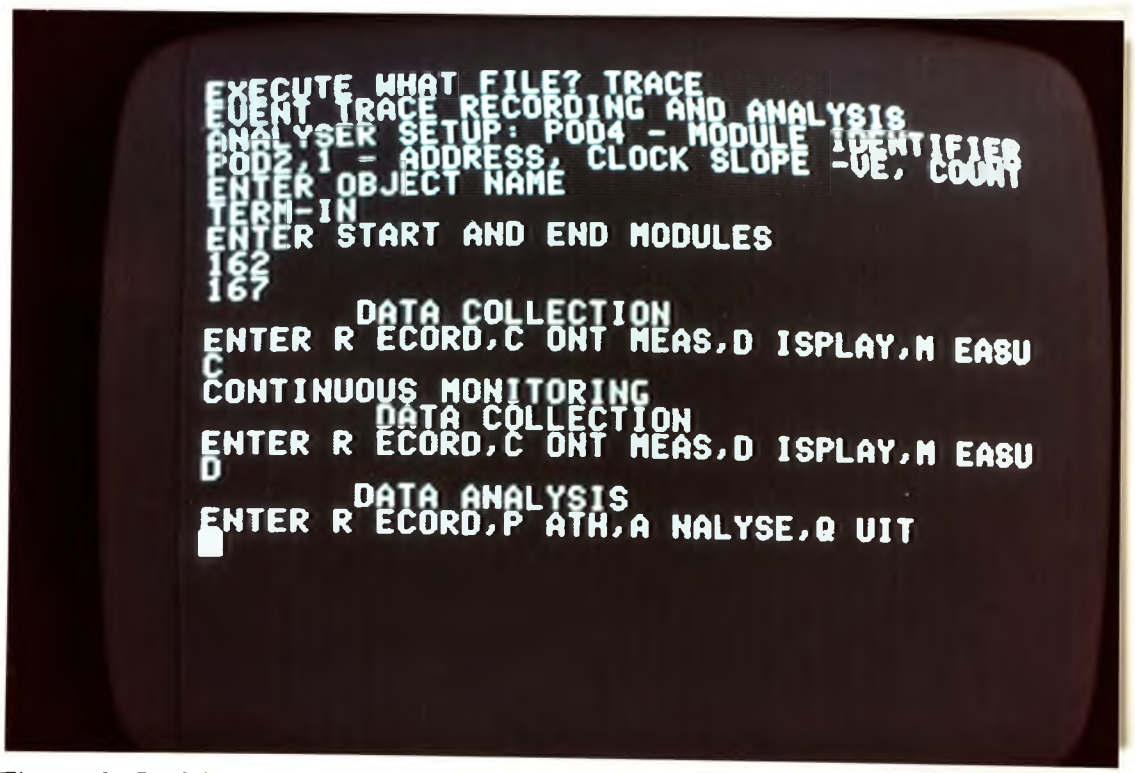


Figure 6.15 Menu Display for Measurement Program (section 13.1) used to select the Terminal Administrator Object (figure 2.4).

Figure 6.16 Object Record, for the ~~Terminmal~~ Terminal Administrator (figure 2.4), produced by the Hybrid Monitoring-Tool - procedure genorec (figure 6.13) - times in microseconds times ten.

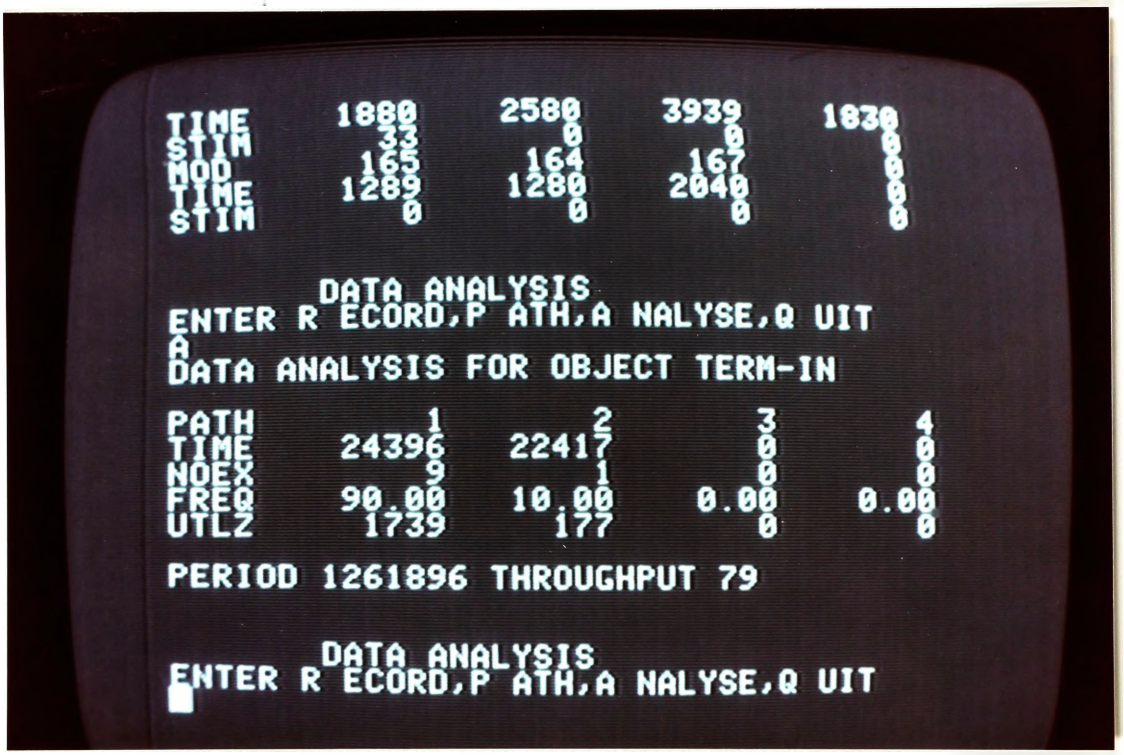


Figure 6.17 Values Calculated from the Object Record (figure 6.16) by the Hybrid Monitoring-Tool - procedure analdat (section 13.1).

Event filters are set up from the state-analysers keyboard. The measurement software (figure 6.15) starts the recording of the event trace (figure 6.14), monitors the state-analysers trace-buffer, and, when it is full, transfers the trace to the Apple II, and then restarts event recording. This sequence is continued until the experiment is halted from the Apple's keyboard.

During the time when the trace is being transferred, no monitoring takes place, so the trace is discontinuous. To minimise the impact of these discontinuities, algorithm 6.4 had to be modified (figure 6.13). The trace buffer is only sixty-four words long. Only object execution-paths which are fully recorded within the trace buffer are recorded; fragments of objects are discarded. The measurement period is the sum of the periods during which the analyser was actually tracing. Thus, this hybrid tool may not record all object executions during the period of the measurement experiment. Also, it could be subject to aliasing if an event triggered several executions of the object and only the first few were recorded in the limited trace memory.

When measurement is terminated, the software enables the analyst to look at a number of tabular displays, or resume measurement. The analyst can look at the complete object record (figure 6.16), at selected path records, at spectrum data, or at calculated values (figure 6.17). This implementation only records the object record to the path-record level. Individual path traversals are not recorded, but some stimulus information is. Graphical displays are hand drawn from the information contained in the object record.

This tool has been used for debugging software, for system development, and for performance measurement (section 8.4). It has proved to be easy-to-use, has saved a lot of time in debugging situations, and its use has increased our understanding of the systems that we have used it on.

7. Monitoring Program Execution

Personal computers are being used for an increasingly diverse range of applications: computer aided instruction in schools, accounting systems in small businesses, bible translation on remote mission stations, etc. However, manufacturers of these systems seem to place more emphasis on games software, as apparently games sell computers, than on program debugging aids. Lack of debugging aids, often accompanied by inappropriate documentation, considerably increases the development time of software, and the frustration of programmers.

Parallel to this revolution in the use of personal computers is an exploding use of microprocessors in manufactured goods: process controllers, instruments, household appliances, cars, etc. The software for these systems is normally developed on a host system and down-line-loaded into the target system. Host systems range from the in-house computer to general-purpose microprocessor-development systems. The latter; through the inclusion of sophisticated, in-circuit emulators; place powerful hardware-for-execution-monitoring in the hands of those who can afford them. The former often provide little more than a monitor, in the target system, which can set breakpoints, examine registers and single step through machine code. Debugging is often complicated by undetected faults in prototype hardware, and by poor understanding of the software/hardware interrelations in the target system.

In recent years, most research effort has been in the area of good design methodologies with the aim of producing programs which work correctly the first time, and hence, require little debugging effort. Plattner and Nievergelt (1981), in their survey of the field, state: *Program execution monitoring has been neglected as a research topic ... [, and] ... program execution monitoring has not kept up with the rapid progress of programming languages.* Glass (1980) has described the development of software for real-time computing as *the "lost world" of software debugging and testing.*

In this chapter, one approach to program-execution monitoring is discussed. The hybrid

monitoring-tool, discussed in section 6.6, has been used to measure program execution at the block-structure level. A case study, from the development of the monitor, is included to illustrate a variety of debugging techniques.

7.1. Programming Tools

Tools, designed to assist the mental activities of the programmer, have been developed for all stages of the programming process. Design tools are used to divide a problem up into intellectually manageable portions, to devise a structured solution to the problem, and to express the solution in a graphical or written form. Coding tools are used to convert the design into a machine understandable form. Up to this point in the programming process, testing and debugging consist of a static analysis of the program, and design, text. Consistent application of these tools, combined with sufficient understanding of the problem and a working knowledge of the implementation language, should result in a 'nearly' correct program.

Next, in the dynamic-program-analysis phase, the program is executed with selected test data, the program produces results (both expected and unexpected), and the programmer studies the execution history of the program to determine the correctness of the program. If the execution history is unavailable the programmer is forced back to static analysis of the text.

Tools which are designed to provide either a snap shot of the program at a particular point in its execution or a complete execution history are called program-execution monitors. Simple execution-monitors are the insertion of breakpoints in machine code and the addition of write statements to program source code (Huang 1980).

7.2. Program Execution Monitoring

The fundamental question faced by both the programmer and the designer of a program-execution monitor is: What information should the monitor capture if it is to be a useful tool. This question can be refined into two further questions: What questions can be asked about the execution of a program, and What properties should a program execution monitor have if it is

to answer these questions. The answers to these questions are found by applying the formulation of performance measurement at the program-execution level of the object hierarchy. The questions to be asked about an object being monitored will depend upon the function of the object and the context in which the object is found.

At the program-execution-monitoring level of the performance-evaluation hierarchy; an object is a whole program, or a block of code; the function of the object depends upon the problem it is designed to solve; and context includes: the mode of execution of the object (stand alone, interactive, etc.), and any programs or events which call for the object to be executed. For example, an object might be a mailing-list program, or a search procedure called by that program. The program object is decomposed into block-structure modules.

However, even though the questions to be asked about a particular program depend upon its function and context, these questions can be generalised to a standard set which apply to all programs, of any function, in any context. At the program-execution-monitoring level of the object hierarchy, the measures defined in the formulation of performance measurement (section 2.6) provide answers to the following questions:

- In what order were the statements (or blocks of statements) in the program executed?
- How long did the program, or a section of it, take to execute?
- How often is the program, or a section of it, executed?
- Does the program spend an excessive amount of time in one section?
- What information gives meaning to the action of the program, e.g. what condition causes it to loop?

The hybrid monitoring tool discussed section 6.6 has been designed to collect the information needed to answer these questions. It is usable on any computer system, once a simple hardware probe has been inserted into that system and software probes have been added to the target program. The tool collects and analyses program execution data on-line in response to interactive

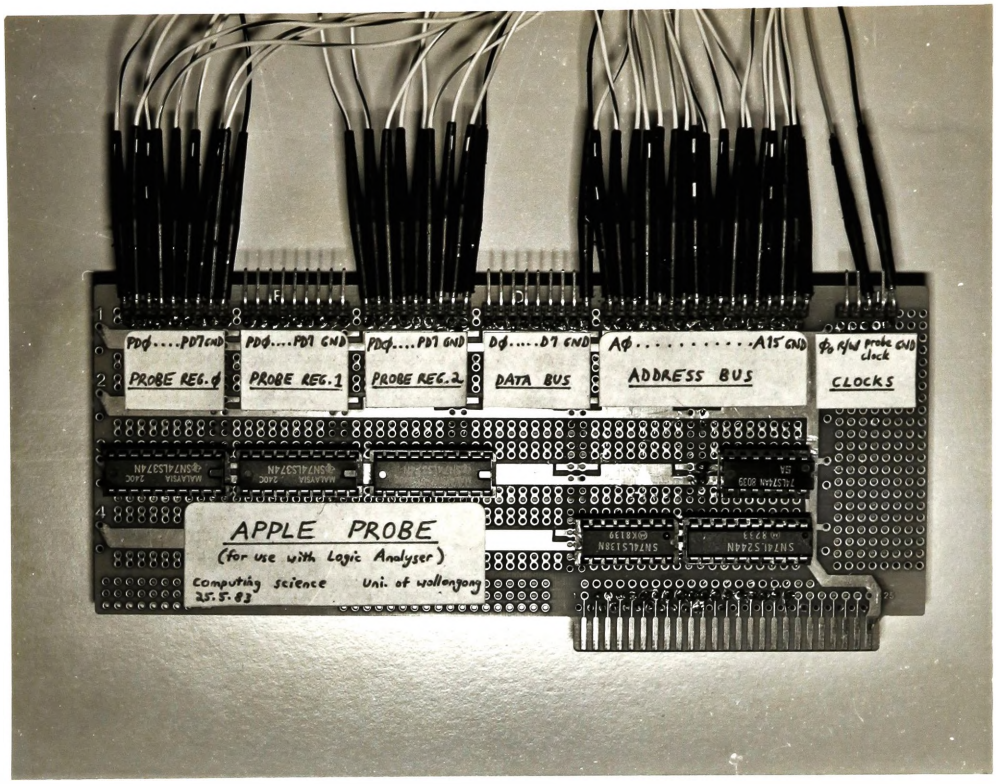


Figure 7.1 Apple II Hardware Probe with logic analyser connectors pushed onto probe pins.

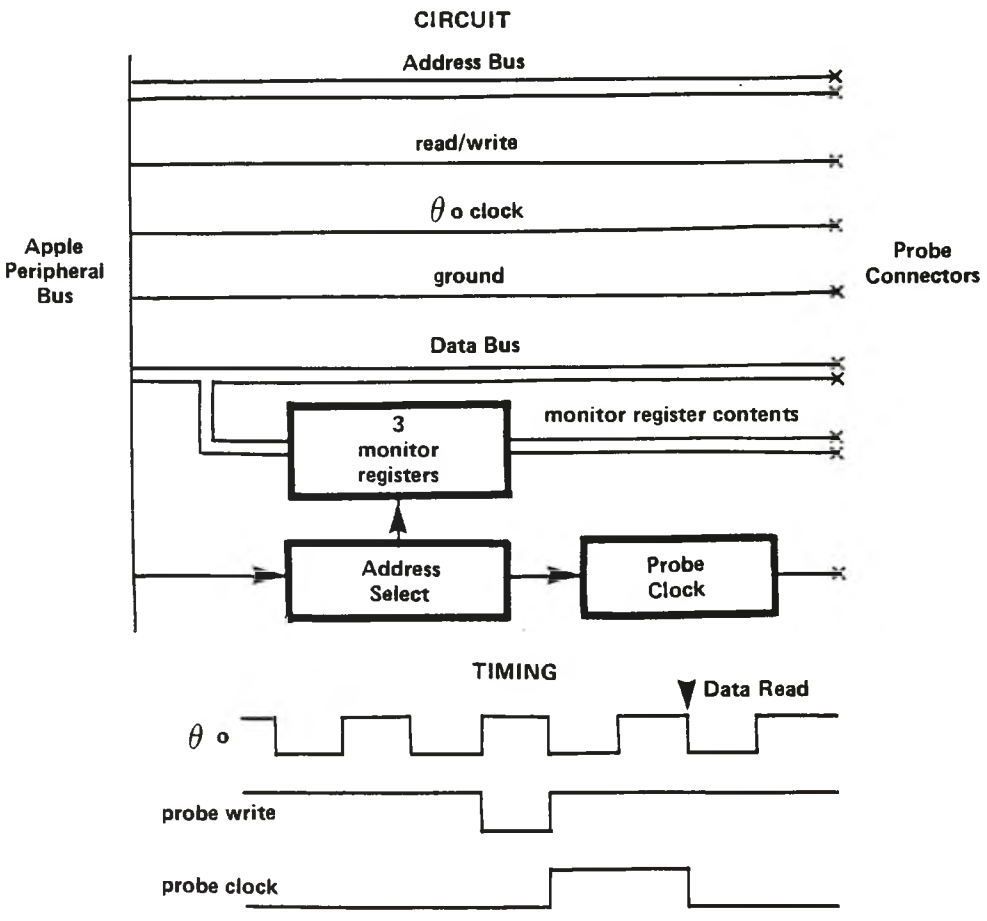


Figure 7.2 Apple II Hardware Probe, Block Diagram

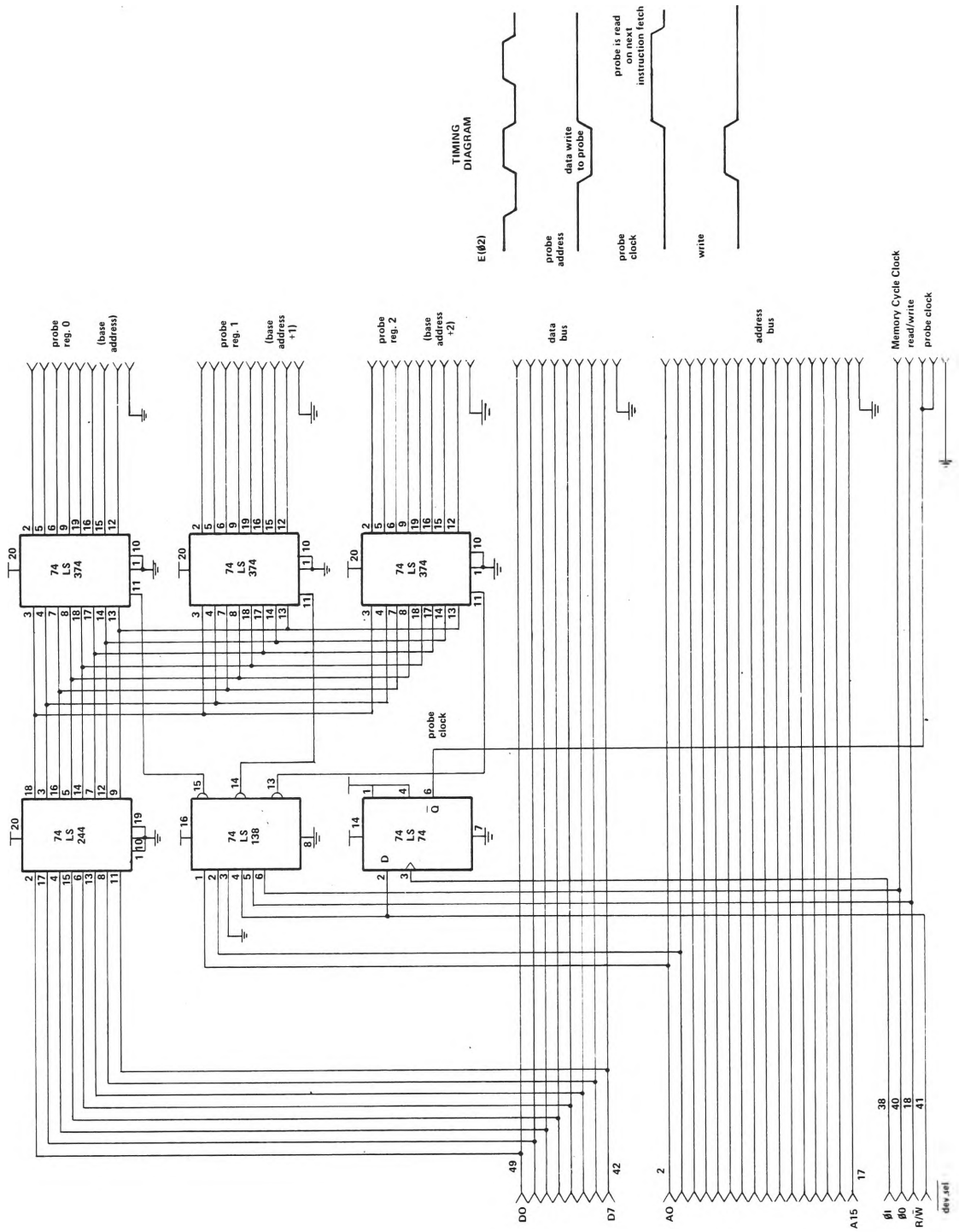


Figure 7.3 Apple II Hardware Probe, Circuit Diagram

user-commands.

7.3. Hybrid Monitoring Tool

Approaches taken to the design of program execution monitors are the same as those used in performance evaluation (see chapter 4 for a detailed discussion of these). Some have implemented software tools, for example Grishman's debugging language (1971); others have implemented hardware tools, for example Fryers memory-bus monitor (1973). Most of these tools, with the exception of microprocessor-development systems, have been designed for specific applications, thus, these tools are not available for general use.

The intergration of hardware techniques and software techniques in hybrid tools gives them the flexibility to handle situations where purely hardware tools or software tools are inadequate. When measuring some programs both software specific information and hardware specific information are needed to understand what the program is doing, for example: measuring the execution time of a keyboard handler processing a variety of inputs. Other programs include some sections where hardware techniques are easier to use, for example: an assembler procedure stored in read only memory, together with sections where software techniques are easier to use, for example: measurement of a high-level language program at the block-structure level.

The hybrid, program execution monitor (figure 6.12) consists of:

- software probes inserted into the target process (figure 7.4),
- hardware probes plugged into the target computers back-plane (figures 7.1 - 7.3),
- a logic-state analyser, which monitors and records the signals from the hardware probes (figure 5.1), and
- a monitoring computer, which controls the measurements and analyses the collected data.

7.3.1. Logic-State Analyser

Extensive data-reduction facilities, and high-speed trigger-capabilities, found in the logic-state analyser are used to initiate program-execution traces, when a specified data pattern occurs, and to detect the states which are to be recorded. Arming of the analyser can be done from the monitoring computer as well as from the analyser's keyboard. The limitations of logic-state analysers as hardware monitoring-tools are discussed in chapter 5 (McKerrow 1983). Even though the latest generation of logic-state analysers (see Corson (1983) for a description of one of these tools) contain very powerful, software-performance, analysis features they still suffer from the limitations of traditional hardware tools. These features are difficult to use on any language other than assembler, and cannot be used if a memory map is not available. A static memory map of the program may not be adequate either, because the data space may be dynamic. The following comment, taken from a personal computer manual, indicates the difficulties to be faced when using a hardware tool:

> *The memory map ... is provided for your curiosity only: a primary task of the transportable Apple Pascal system is to eliminate the necessity for the programmer to know anything about specific memory addresses and use.* (Apple 1980, page 254).

7.3.2. Hardware Probe

The concept of a monitor register, where all signals of interest are multiplexed through a single register, was introduced, by Deese (1974), to overcome some of the problems inherent in connecting a hardware monitor to a computer. In the hybrid monitor, multiplexing is done in software, and the hardware probe (figures 7.1 - 7.3) contains monitor registers and bus connectors. Bus connectors connect the processor's bus-signals to the logic-state analyser for monitoring memory accesses, machine-code execution paths, and data transfers. Monitor registers provide the interface between the software probes (routines inserted into the program under study) and the logic state analyser. When an event of interest occurs a software probe writes information about that event to a monitor register, from where the information is read by the logic-

Assembler Probe

```
ADDR EQU 0C0D0 ; PROBE ADDRESS
    LDA #02   ; MODULE IDENTIFIER
    STA ADDR   ;WRITE TO PROBE
```

Assembler Poke Procedure

```
ADDR := -16176; (*PROBE IS IN SLOT 5*)
POKE(2,ADDR);

      .PROC POKE,2
; PROCEDURE POKE(DATA,ADDR:INTEGER)
;PROCEDURE TO WRITE TO ADDRESS
RETURN .EQU 0
ADDR   .EQU 2
      POP RETURN   ;SAVE RETURN ADDRESS
      POP ADDR     ;MEMORY LOCATION
      LDX #0
      PLA          ;GET OUTPUT
      STA @ADDR,X  ;POKE
      PLA          ;CLEAN UP STACK
      PUSH RETURN
      RTS          ;GO BACK
```

Pascal Variant

```
TYPE MAGIC = RECORD
          CASE BOOLEAN OF
          TRUE : (INT:INTEGER);
          FALSE : (PTR:^PA);
          END;
VAR CHEAT:MAGIC;

CHEAT.INT := ADDR;
CHEAT.PTR^[0] := 2;
```

Figure 7.4 Variety of Software Module-Identifier-Probes - Instrumented program in section 13.2

Implementation		Time in microseconds		
Language	Code	Before	After	Total
Assembler	two instructions	6	0	6
Assembler procedure	poke(data,address)	427	25	454
Pascal variant	probe.ptr^[0]:=data	609	12	621

Table 7.1 Cost of Software Probe - on an Apple II and UCSD Pascal - probe code given in figure 7.4

state analyser.

All signals, including the probe clock, are synchronised to the processor's memory-cycle clock. When one of the monitor registers is updated the probe clock is pulsed, with its active edge occurring at the end of next memory-cycle, consequently, the analyser's reading is synchronised with an instruction-fetch memory-cycle. This synchronisation is important when monitoring microprocessors, where no external signals are available to distinguish instruction-fetch memory-cycles from other memory-cycles.

Two types of data are presented at the hardware probe connectors: the contents of the monitor registers, and the following bus signals: memory address, memory data, read/write signal, and other available control signals if desired. From the bus information, the monitor can determine what the computer is doing and record its execution path at the machine-code level.

7.3.3. Software Probes

Determining why a program has taken a particular path, and recording the execution path of programs at the block-structure level, cannot be done with a hardware monitor. Information stored in the monitor registers, by the software probes, can be used to determine both the execution path of an program, and the conditions under which particular paths are taken. Probe software (figure 7.4) increases the execution time of the program under study, but the execution time of the probe software can be measured (table 7.1) and correct timing calculated. Once the execution history of a program is known the probe software can be removed.

Monitor registers are used to store two types of information: module number, and stimulus data. Each program can be divided into one or more modules. A **module** is a contiguous piece of code that is executed in response to an event, for example a procedure or a compound statement. An **event** is any action that initiates a significant change in the state of the program, for example a procedure call. Each module has a unique module-number which is written to the module-number monitor-register by a software probe at the start of the module. By monitoring this information the analyser can record: the execution path of the program at

the block-structure level, the time taken to execute each module, and the start address of each module (providing a memory map of the modules for lower-level monitoring). Thus, programs written in high-level languages can be monitored at the block-structure level without knowledge of the machine code or memory map.

Module-number software-probes have been implemented in two ways: a simple write to the module-number monitor-register (figure 7.4); and a more complex method, involving the maintenance of module numbers on a stack (section 8.4), in situations where the execution of modules could be interrupted. The latter method costs more in execution time, but simplifies implementation in complex objects where there may be several levels of procedure nesting. In both methods of probe implementation, the position of the instruction which writes data to the monitor register (probe-write instruction) within the software probe is important. The logic-state analyser reads the contents of the monitor registers, etc, at the end of the next memory cycle after the probe write. This memory cycle is an instruction fetch, and thus, the address read from the hardware probe is the address of the next instruction.

If the probe-write instruction is the last instruction in the probe software (see assembler probe in figure 7.4) then the address read from the hardware probe is a target-program address. If the probe-write instruction is not the last instruction in the software probe (see assembler poke procedure in figure 7.4) then the address read from the hardware probe is not in the target program, and is of no use to the analyst. One way to guarantee that the address read from the hardware probe is a target-program address is to use hand-crafted macro routines instead of subroutines (figure 8.5), if the implementation language will allow it. Using macros will give target-program memory-map-data for both methods of probe implementation, for the cost of increased memory usage by the probe software. Having target-program addresses available, in addition to module identifiers, is of considerable advantage when debugging programs, and when running monitor verification programs.

Stimulus information; data indicating why a program is taking a particular path, or any other data the programmer wishes to look at; can be written to the stimulus-probe monitor-

```
    'T' : BEGIN (* APPLE TO TALK*)
          POKE(3,ADDR); (* module identifier probe *)
          OUTPUT(TALK,5); (* talk statement module figure 7.6 *)
          END;

PROCEDURE OUTPUT(VAR STR:STRNG;N:INTEGER); (* SENDS A STRING TO IEEEBUS *)
VAR I,STIM:INTEGER;
   DAT:CHAR;
BEGIN
POKE(8,ADDR); (* character output procedure figure 7.6 *)
FOR I := 1 TO N DO
  BEGIN
  DAT := STR[I];
  STIM := ORD(DAT);
  POKE(STIM,ADDR + 2);
  PUTCHAR(DAT);
  END;
POKE(7,ADDR); (* return to talk statement figure 7.6 *)
END;

      .PROC PUTCHAR,1  ; PROCEDURE PUTCHAR(DOUT:CHAR);
      .PUBLIC PSTATUS,PHZWD,PPOSN,POUT,PIN,PDELM,PCCLN
RETURN .EQU 0
DATA   .EQU 2
STAT   .EQU 7
HZWD   .EQU 21
POSN   .EQU 24
OPHK   .EQU 36
INHK   .EQU 38
XREG   .EQU 46
DELM   .EQU 5FB
CCLN   .EQU 7FB
ADDR   .EQU 0C0D0
OUTPUT .EQU 0C32B
      POP RETURN
      POP DATA
      LDA #0F0
      STA ADDR ; first save restore module figure 7.6
      PUSH OPHK
      PUSH INHK
      PUSH XREG
      PUSH DELM
      PUSH CCLN
      MOVE PSTATUS,STAT
      MOVEB PHZWD,HZWD
      MOVEB PPOSN,POSN
      MOVE  POUT,OPHK
      MOVE  PIN,INHK
      MOVEB PDELM,DELM
      MOVEB PCCLN,CCLN
      LDA #0F1
      STA ADDR ; firmware transmit module figure 7.6
      LDA   DATA
      JSR OUTPUT  ;write a character to IEEE bus
      LDA #0F0
      STA ADDR ; second save restore module figure 7.6
      MOVE STAT,PSTATUS
      MOVEB HZWD,PHZWD
      MOVEB POSN,PPOSN
      MOVEB DELM,PDELM
      MOVEB CCLN,PCCLN
      POP CCLN
      POP DELM
      POP XREG
      POP INHK
      POP OPHK
      LDA #08
      STA ADDR ; return to character output procedure figure 7.6
      PUSH RETURN
      RTS
```

Figure 7.5 Instrumented Test Program - complete program in section 13.2

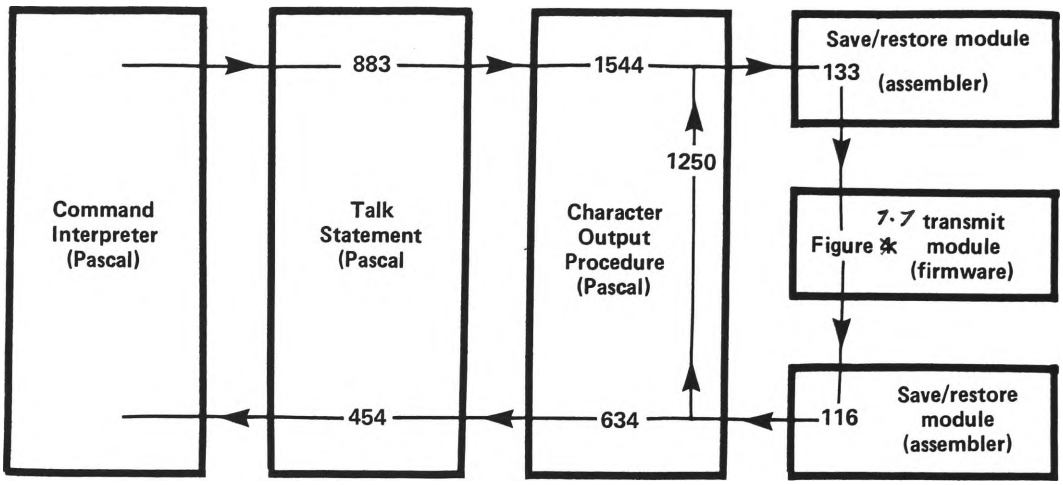


Figure 7.6 Execution-path flow-graph for the program in figure 7.5 showing the flow of program execution when the Apple is talking (times in microseconds)

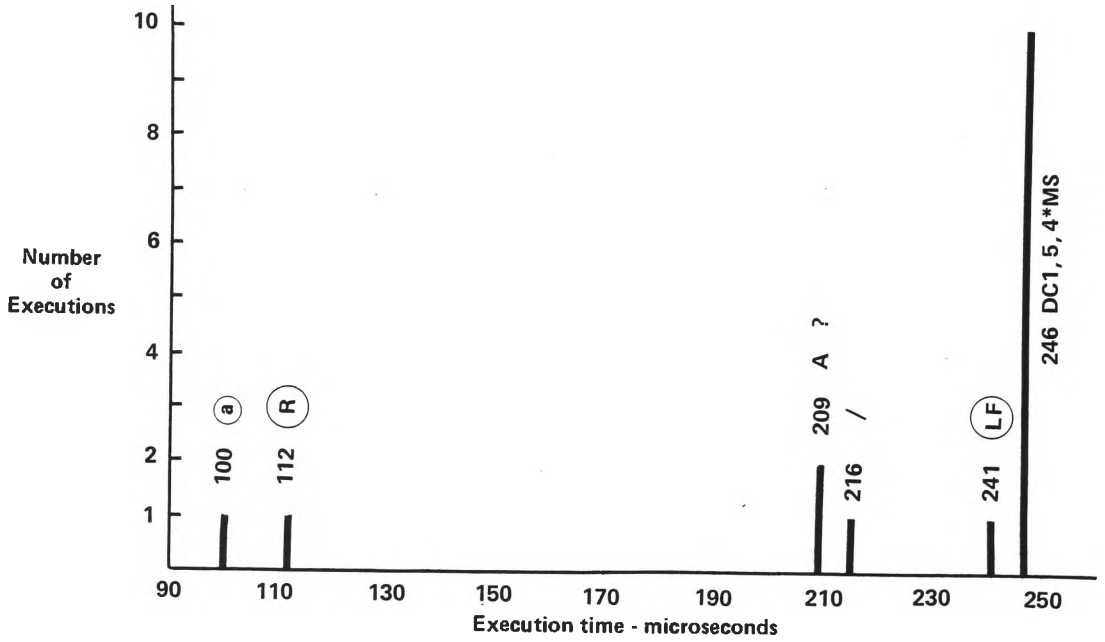


Figure 7.7 Module Spectrum of the Firmware Transmit Routine in the IEEE 488 Interface,
for the command string: @?A/ R
followed by the data string: DC1,5,4*MS LF

register at any time. An object **module-map**; which includes: module name, module number, module stimulus-information, and the order in which stimulus information is written; must be documented when the probes are inserted into the software.

7.4. Program Execution History

Data collected by the logic-state analyser is analysed, by programs in the monitoring computer (Apple II), to find answers to the questions raised in section 7.3. Prior to making any measurements, the programmer decides what he wants to know about the program under study. Then he arms the logic-state-analyser's triggering-circuits, to detect the state at which the recording of execution history is to start, and he sets up the logic-state-analyser's data-reduction circuits, to record only data of interest. As the target process executes, data written to the hardware probe is recorded until either the target process terminates or the analyser's memory is full. At the completion of the measurement, this data, which is the execution history of the object under study, is transferred to the monitoring computer for analysis. Recorded data is also displayed on the logic-state-analyser's screen allowing manual analysis.

One program studied with the program-execution monitor was the program used to develop the Apple-to-Analyser communication link (section 13.2). The Apple IEEE488 bus interface included a driver in read-only memory. Calling this firmware routine from Pascal programs proved to be very difficult. Consequently, the hybrid performance-measurement-tool was used as a program-execution monitor to debug the software (section 7.5), and to measure what the firmware was doing.

The **execution path** of part of the program (figure 7.5), and the time each module took to execute, can be read directly from the execution-history, and then drawn as a flow graph (figure 7.6). Modules, which have several execution paths, can be characterised by plotting a module spectrum. A **module spectrum** (figure 7.7) was produced, by plotting the time taken to execute the module versus the number of module executions which take each path, for the firmware routine which handles output to the IEEE bus and sets up the mode of operation of the inter-

face card. As there is a discrete number of paths through a module the spectrum consists of a vertical line for each execution path. Stimulus information, also recorded on the spectrum, defines the conditions under which different paths are executed. The most frequently used execution paths, and execution paths which take excessive amounts of time, can be read directly off the spectrum. For this program (figure 7.7), the data-output-to-the-bus path is the most frequently used path, and is the path which has the longest execution-time.

Full characterisation of a module is possible only if all execution paths are taken. Monitoring the execution paths of a module at the machine-code level, for all known execution paths, will show up any areas of the program address space that are not used, and thus, indicate the presence of additional execution paths. Alternatively, at the block-structure level, comparing the modules in the event trace to the module map will indicate any blocks that are not executed.

When monitoring execution-paths at the machine-code level (section 5.5.1 for a discussion of machine-code-level monitoring on a minicomputer), a method of isolating instruction-fetch memory-cycles from the larger set of memory-cycles is needed. On minicomputers, signals indicating the type of memory cycle (instruction fetch, operand fetch, data fetch, and data write) can usually be found. Unfortunately these signals are internal to most microprocessors, and as a result, data reduction has to be done by one or several of the following methods: separating data and address space and recording only address-space cycles, comparing the execution history with a machine code listing, or emulating the processor (as is done in microprocessor development systems).

Monitoring the execution path of a module, written in a high-level language, at the program-instruction level can be done by inserting stimulus probes between program statements. The cost, and tedium, of this exercise is such that it would only be considered as a last resort in program debugging, or as a means of determining the execution time of individual high level instructions.

7.5. Program Debugging

To illustrate the techniques developed in the previous sections, the debugging of the IEEE488 communications program (section 13.2), in the monitoring computer, is described below. The documentation for the IEEE488 interface card (Apple 1979), which was bought for the Apple II, included the following specification:

> *The IE-01-79 [interface card] comes with on board firmware (listing in Appendix D) and requires no additional machine code software or memory space to operate. The firmware was written to enable the user to invoke communications with the IEEE bus from a high level language and programming examples presented in this manual are in Applesoft or integer basic.*

7.5.1. Traditional Methods

Several Basic programs, written to test the operation of the card, worked in all but one mode. For a number of reasons the analysis software was to be written in Pascal, but attempts to drive the interface from Pascal failed because the card was designed to be driven from Basic. Pascal and Basic handle input/output in a different way: Basic has only one input device and one output device, requiring only one input and one output vector; Pascal can have a number of input and output devices each with its own vector. The differences, including the bank switching of interface firmware (firmware on all interface cards resides at the same address), were compensated for with an assembler routine. The program appeared to initialise the IEEE bus, with Apple to talk and Analyser to listen, but as soon as data transfer was attempted the program crashed and the Apple rebooted.

Examining the listing of the program revealed that the firmware was using additional memory space for the storage of variables. The memory locations used were valid temporary locations in Basic but not in Pascal. Assembler routines were added before and after the calls to the interface card firmware to save and restore these memory locations (figure 7.5), but the fault remained. Using the disassembler, in the Apple monitor, the firmware was disassembled

and found to be different to the listing: one mode of operation had been left out completely, and minor changes had been made to the rest of the code.

7.5.2. Hybrid Methods

At this stage a program execution monitor was needed, so the hardware probe was plugged into the Apple and software probes were inserted into the program. The stimulus information chosen was the actual character being written to the Interface card by the Pascal program. Then the execution path, through the firmware, for each character written was monitored, and an execution history built up (figure 7.6). The Pascal code was monitored at the block-structure level, but because probes could not be inserted into firmware, the firmware was monitored at the machine-code level. The firmware could be monitored as a block, but it could not be decomposed into block-structure level modules. Dynamic analysis of the program was considerably faster than static analysis of the code, which had proved to be difficult due to the complexity of the code.

From the execution history, the following were determined: the interface was being set up correctly; the interface was switching from command mode to data transmission mode, when the terminator of the command string was processed; the first data character was being transmitted, but the execution path included a routine which deselected the firmware memory bank; and, consequently, the crash occurred, during the attempt to transmit the second character, when a call to the firmware routine jumped to a non-existent memory location and fetched the code for a software interrupt, which caused the Apple to reboot. This problem was overcome by reselecting the firmware memory bank before the firmware was called.

Some confusion, when examining the firmware execution history, was caused by the fact that during the processing of a return-from-subroutine instruction, by the 6502 processor, the address of the next memory location appears on the bus for one memory cycle. Often this address is the start of another subroutine indicating, falsely, that the subroutine had been called.

Now the program could transmit the text of the message over the IEEE bus, but when the termination character (a carriage return) was processed the Apple did strange things and had to be rebooted. Monitoring the execution path of the termination character revealed that the firmware called a routine in the Apple monitor, but when Pascal is running on the Apple a completely different monitor resides in that area of memory. This fault was rectified by using a different termination character (line-feed) whose execution path did not include the monitor call, and then the program could transmit data successfully.

Next, an attempt was made to receive data from the IEEE488 bus. The interface was initialised correctly, but as soon as data transfer was attempted the Apple appeared to die. Monitoring the execution path revealed that the program had in fact hung up in an infinite loop. Comparing the execution path to the listing showed that the path should not have been possible, because, the program was branching on the state of a variable to a routine which reset that variable. Obviously the variable was being modified elsewhere. Writing the variable to the stimulus probe confirmed that it was being modified in the save/restore routine between the Pascal host and the firmware. This module worked in the call to the output firmware-routine, so why didn't it work in the call to the input firmware-routine.

The save/restore routine uses global variables declared in the Pascal host, and references to these variables are resolved when the assembler routine is linked to the Pascal host. The linker had not reported any errors, and the documentation (page 181 Apple 1980) does not explain how to interpret the information in the link map. A linkage fault was confirmed by monitoring the execution path, and the data-space accesses, of the save/restore module in both the working and the non-working cases. Changing the order of the assembler procedures, in the library of routines to be linked in, overcame the fault (a rather unusual fix), and the program could now receive data but failed to see the termination character. Writing the received characters to the stimulus probe revealed that the most-significant bit of the incoming characters had been set causing the Pascal equality test to fail. This problem, and others caused by the data being reduced from 8 bits to 7 bits, was overcome by replacing the firmware input-routine with

a modified copy stored in random-access memory (GETCHAR in section 13.2). The modified copy used permanently allocated variables, and hence, the save/restore routines were eliminated.

7.6. Conclusion

During the development of a program, there are many occasions where the process of identifying program faults can be simplified by examining program execution-histories. However, program execution-monitoring has been a neglected research topic, consequently, few general purpose tools are available for dynamic program-analysis.

A hybrid program-execution-monitor, built from readily available components, has proved to be a powerful, easy-to-use, dynamic program-analyser. Only minor additions are required to the hardware of the target computer, and the cost of the software probes can be measured, eliminating errors in timing measurements. A method of measuring program execution histories at the block-structure level, which has been introduced, removes any need for the programmer to know details of the machine code or the program memory-map. Thus, the methodology is consistent with modern philosophies of programming in contrast to many techniques in common use, which require an intimate knowledge of low-level implementation details. Low-level details can be measured by the tool, if required.

Logic-state analysers are currently more expensive than personal computers, however, an Oregon Company has just announced a logic-state analyser which is a peripheral to an Apple II computer (section 5.6). If this instrument is successful, it will stimulate the development of low-cost, personal-computer-based, logic-state analysers placing powerful, dynamic-program-analysis tools in the hands of many programmers.

8. Computer System Design for Measurement

Increasingly, computer users are faced with black boxes built without regard for the user's need for information about system operation. In many ways computers complement human intelligence, but if a computer fails to provide information that is easy for the user to assimilate, confidence in the computer is diminished.

Often, when executing a copy command on a dual floppy-disc system, one wonders if the copy is going the correct way. Lights indicating reading and writing operations would give the user increased confidence through more rapid feedback. Computers are excellent at handling numbers and tables; humans at recognizing patterns and pictures. The removal of speakers and front panels from computers have made them less friendly.

The operating system used in a real-time control project (McKerrow 1978) was modified to illuminate an individual light on the console for each process, when that process was active. As many of the processes were cyclic, due to the nature of the external machines, a regular pattern could be observed on the lights when the system was operating correctly. This enabled the operation of a complex control system to be analysed visually. On a number of occasions, control system problems were diagnosed by an engineer using information about the light pattern, relayed over the phone by electricians who subsequently fixed the faults.

Simple tools like these may not provide very accurate measures, but they do inform the user about what is happening. Some new systems have an icon moving about the screen during program execution to inform the user that something is happening, i.e. the system has not died. These tools all increase user confidence, and should be included at design time. If they are not, they can be difficult to add later.

The majority of monitoring tools are designed to monitor existing systems. Consequently, serious problems, due to the constraints imposed by the organization of the system being monitored, are often encountered. These problems include:

- Events of interest are not accessible;
- Excessive interference;
- Difficulty in verifying collected data;
- Modifications to the system to provide the required data may be difficult, risky or expensive; and
- It may not be possible to place the probes where you want them.

In addition to making measurement difficult, these problems can cause errors in the measurements.

These problems could be minimised if a comprehensive set of measurement facilities were included during system design. This practice has not been popular so far among computer manufacturers, even though such tools would then be available for use during system implementation and debugging. Research projects into tool integration include the instrumentation of Multics (Saltzer and Gintell 1970) and of PRIME (Ferrari 1973). One of the main problems faced by the designers of these systems was: how do you predict which events will be of interest for measurement purposes once the system is implemented. As a result, both projects adopted a mixture of ad-hoc fixed tools and general-purpose tools.

To be effective in the long term, performance evaluation has to be considered when the system, both hardware and software, is designed. We should no longer design hardware, software and instrumentation separately and then patch until they work. An integrated approach is necessary. Before such an integrated approached can be achieved, performance evaluators will have to be able to specify more clearly, at design time, what they want to measure and why.

Only a few systems have been adequately instrumented by their designers. One of the problems of incorporating tools into a system during design is that it is difficult to determine measurement needs when the system's specification has not been stabilised. Ferrari (1973) comments: ... *in the absence of a general theory of performance evaluation, the only way to over-*

come this difficulty is to build into the system general purpose tools with sufficient power and flexibility as to allow the system's evaluators to measure practically any variable they may be interested in. One of the corollaries of the formulation of performance measurement is a methodology for including performance instrumentation at design time.

To apply the performance measurement methodology (chapter 6) to the instrumentation of a system the designer must understand the system. The decomposition of the system into tasks, and tasks into modules, will vary from system to system depending upon the design of the kernel. The operating system designer should understand the system better than anyone else. Hence, he should be better equipped to instrument it than anyone else. Thinking about instrumentation will force the designer to implement the code in a structured way.

Once the operating system design is realised, it can be measured to see if it is operating in accordance with the design. Thus, immediate feedback on the operation of the system is available to the designer, who can then test improvement hypotheses. Also, measurements made on one system can be used to build models during the design of new systems (Lynch 1972). In this way, designers can improve the design, and performance of operating systems from one generation to the next.

Another goal of operating system designers is adaptive control of system performance (Boulay et al 1977, Geck 1979, Serazzi 1981). Feedback control requires integrated instrumentation, coupled with an understanding of the impact of parameter changes on performance. This understanding is gained by changing parameters and measuring the resultant performance change. From these measurements, a transfer function for the system can be found.

8.1. Instrumentation of Multics

Multics was developed in a research project whose intent was to create an operating system, centred around the ability to share information in a controlled way, which could support a wide variety of computational jobs (Saltzer and Gintell 1970). A spectrum of user services; including a hierarchical file-organisation, sharing of information in core memory, dynamic linking of sub-

routines and data, parallel processing, and device-independent input/output facilities; characterises the system, and contributes to a complexity that makes careful instrumentation mandatory. Two features of Multics that were of particular interest to analysts were multiprogramming and demand paging.

Part of the Multics research project was to try out new ideas, and new combinations of old ideas. As a result, a large number of design choices had to be made between different algorithms, strategies, parameter settings, and algorithm implementations. It was presumed, from the start, that some wrong choices would be made, so there was an emphasis on integrated instrumentation. The result was an ability to recognise bottlenecks. Two effects were observed:

- frequently, the best guesses by system programmers as to the cause of a performance problem were proven to be wrong by detailed measurement, and
- many otherwise undetected performance problems were discovered while studying measurements.

It was found that performance degradation of the order of twenty per cent regularly goes unnoticed by users. Thus, integrated instrumentation has two significant advantages:

- programming effort is reduced, because incorrect hypotheses are spotted, and programmers do not spend time optimising the wrong code module, and
- performance problems which are not perceptible to the user can be detected.

The instrumentation included in Multics was directed primarily towards the goal of understanding what goes on inside the operating system. Not all the measurement techniques were thought out in advance. A large part of Multics measurement research was the discovery of what measurement facilities were needed. Multics included a combination of hardware, software and hybrid tools.

Three hardware tools came with the GE645 computer on which Multics was implemented: a program-readable clock; a memory-cycle counter; and an externally-driven input/output channel, which enabled an external computer to monitor the contents of the GE645's primary

memory. Associated with the program-readable clock was a programmable comparator, which generated an interrupt whenever a match occured.

Some of the software tools included in Multics performed the tasks described below:

- A general measuring package recorded the time spent executing selectable supervisor modules and their frequency of execution.
- A segment utilization meter sampled, every ten milli-seconds, the segment number of the segment which was executing and stored the result in a table. This provided a simple way of detecting how time spent in the system was distributed among the various components.
- The number of missing pages, and segments, encountered during execution in a segment was recorded on a per segment basis.
- The number of procedure calls was counted.
- The sequence of missing pages encountered by a task was recorded. This record frequently indicated poor locality of reference, and hence a higher than necessary use of system resources by a program.
- The effect of the system's multiprogramming effort on an individual user was traced.
- Feedback was provided to the user about the resource utilization of the command just typed (time of day, cpu use by program, and number of page misses).

Multics did not have a built-in general event tracing package. It was thought that the volume of data, produced by such a package, would be too large to analyse.

A **graphics display monitor** (Grochow 1969) on a separate computer, connected by the channel, included a variety of standard displays which were used to observe the traffic controller's queues, the use of primary memory, and arrays produced by other tools. During system initialization, Multics built a table containing pointers to interesting data bases for use by this hybrid tool. The monitoring computer was also used to generate a simulated work load for measurement experiments. A number of typical user interaction scripts were developed. Having logged on to Multics, the monitoring computer could simulate up to twelve interactive

users using these scripts.

Saltzer and Gintell (1970) comment,... *building permanent instrumentation into key supervisor modules is well worth the effort, since the cost of maintaining well-organized instrumentation is low, and the pay-off in being able to 'look at the meters' any time a performance problem is suspected is very high.* In addition, the presence of instrumentation helps to eliminate the non-scientific approach taken by many programmers to finding bugs. Measurement is always better (provided its accuracy has been verified) than guesswork. In chapter 9, a number of other systems that have been designed for measurement are discussed.

8.2. Design of the MU5

The design of the MU5 at the University of Manchester was based upon measurements of the Atlas computer (Sumner 1974). The Atlas hardware generated interrupts to the operating system on the occurrence of the following events: at the end of every 2048 instruction executions, at the end of every tenth of a second, and at the end of every second. Routines, which executed in response to these interrupts, collected measurement data and stored it into spare fields in the supervisor-log data-structure. A logging program read this data structure regularly and wrote the information to a daily log tape.

To measure the dynamic operand-accesses of high-level-language programs, the compiler was modified to insert a probe whenever an operand was accessed. When executed, the probe incremented a counter. At the termination of program execution, the counter values were printed out. The results of measuring a large number of programs was that 80% of accesses were to variables, and 20% of accesses were to array elements. Further measurements were made to determine the size of data caches for the MU5.

To get a picture of dynamic instruction usage, when the interrupt at the end of every 2048 instructions occurred the currently executing machine-code instruction was classified and an appropriate counter incremented. At the end of the day the counters were read, printed, and analysed. The long duration of the measurement gave a good average over many programs.

The ratio of floating-point-arithmetic-instruction usage to fixed-point-arithmetic-instruction usage, influenced the design of the fixed-point-arithmetic unit of the MU5. A multiplier was designed into the fixed-point-arithmetic unit so that address calculations could be done without having to interfere with the operation of the main arithmetic unit.

They found that 20% of the instructions executed were branch instructions. This explained why the theoretical maximum throughput of the Atlas was not attained. The Atlas had an overlapped architecture where instructions which took seven microseconds had an effective execution time of two microseconds, due to a five microsecond overlap. However, branch instructions did not take advantage of this overlap losing five microseconds of overlap, effectively increasing the average instruction time from two to three microseconds, i.e. a 50% slow down.

To find ways to improve the execution of branch instructions on the MU5, the hardware of the Atlas was modified to generate an interrupt whenever a branch instruction was executed, except in the analysis routine. An analysis routine, called by the interrupt, was used to count the number of instructions executed between branch instructions. A number of histograms were plotted, showing the number of sequential instructions between various types of branch instructions.

Very rarely were sequential blocks, and hence loops, more than ten instructions in length, with a median block length of four instructions. Analysis of the histograms showed that 70% of branch instructions do branch, count-and-test conditional-branch-instructions branch about 80% of the time, and arithmetic-test conditional-branch-instructions branch only 50% of the time.

They also found that on any particular test the branch goes the same way 80% of the time, and 30% of jumps span a distance of less than four instructions. In the MU5, a buffer, which can hold up to eight jumps, is used to store the jump-from address and the jump-to address whenever a branch occurs. When a piece of the program is executed again, the jump-to address stored in the buffer, for this branch, is used as the program counter, while the test is

evaluated. If the branch would not have been taken, the fetched instruction is discarded and another fetched.

Eighty per cent of the time the assumption that the program will branch the same way it did last time is correct and the pipeline runs at full speed. Twenty per cent of the time the assumption is incorrect, and the extraneous bus activity generates a large gap in the pipeline. Careful programming can reduce the number of gaps encountered. For a comprehensive study of branch behaviour on modern machines see Lee and Smiths (1984) article on branch prediction strategies.

Attempts were made to measure the optimum page size for the Atlas, but the right signals were not available. When the MU5 was built, performance measurement tools were included to measure the operation of the pipeline (Yannacopoulos et al 1977). These tools are the same in function as those on the Atlas, but as they were included in the design they have been integrated into the system.

The performance of the branching mechanism was at least as good as expected, but the data cache did not perform as well as expected. This case study illustrates the usefulness of performance measurement in hardware design.

8.3. Microprocessor Design for Measurement

The design of microprocessors involves many compromises between the desirable and the achievable. Higher packing density has enabled the introduction of more powerful architectures but many desirable features are left off. Some have implemented reduced instruction sets in an attempt to free up chip area for use by other circuits. No microprocessor has had performance measurement tools included in it, and compromises on pin count have eliminated a number of useful signals.

If this trend continues, it will become more and more difficult to measure the performance of microcomputer software. What signals should be available for performance measurement? What circuits can be added to a processor to reduce the interference of software tools?

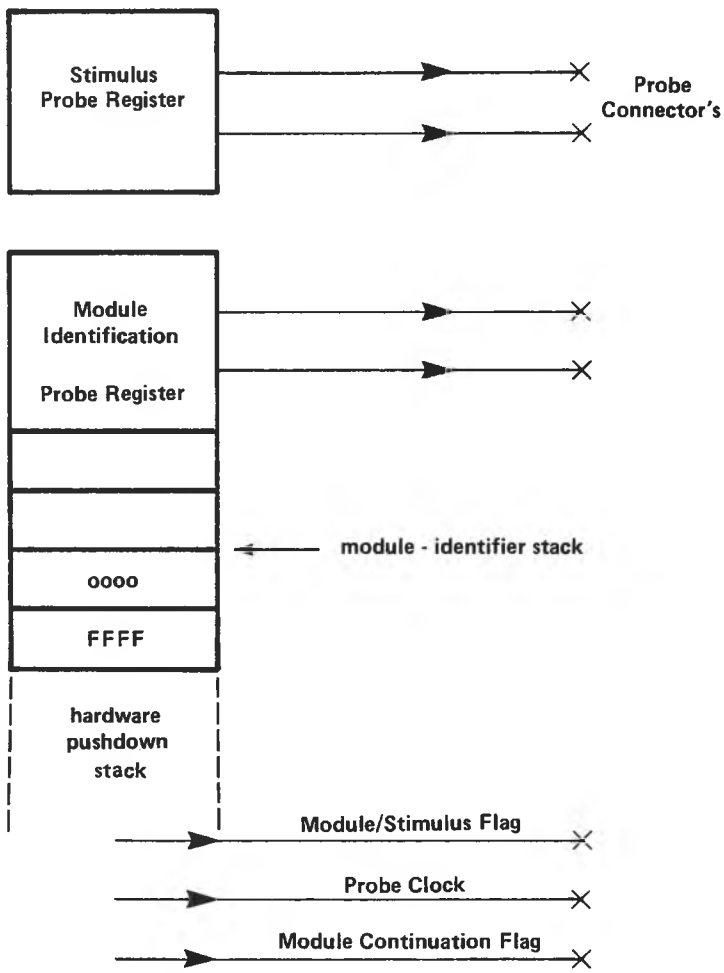


Figure 8.1 Hardware to be added to a microprocessor to reduce probe interference

Opcodes

Probe Writes information to module-identifier monitor-register.
{sets module/stimulus flag, and generates probe clock}
Stim Writes information to stimulus monitor-register
{resets module/stimulus flag, and generates clock}
{sets process and task flags as required}

Probe Instruction Opcode Modifiers

abe - absolute branch entry - A 6.12
- writes new value into module register
{overwrites old value}
- stack is not operated on
- continuation flag reset

inten - interrupt routine entry - A 6.14
- push new value into module register
- continuation flag reset

intex - interrupt routine exit - A 6.15
- pop stack {old value into module register}
- continuation flag set

idle - return to quiescent state - A 6.17
- pop stack until zero in module register
- clock generated after each pop
- continuation flag reset on last pop

Stim Instruction Opcode Modifiers

proc - this is a process number
- write to first stimulus register

task - this is a task identifier
- write to second stimulus register

num - number of stimulus register to write to

Operand

- value
-address of value {normal instruction addressing modes}

Figure 8.2 Probe Machine Instructions - Used in conjunction with Monitor Chip

The signals needed by an external hardware tool all relate to bus information. They include:

- address bus,
- data bus,
- memory cycle code - instruction fetch, operand fetch, data read, data write, dead cycle, interrupt vector, dma cycle - used to qualify the memory cycle clock,
- memory cycle clock,
- user/supervisor mode flag,
- busy/wait flag,
- exception processing flag (i.e. interrupts disabled), and
- processor-interrupt priority-level

Some of these signals are not present on current microprocessors. The most important additional signals are those used to qualify the memory cycle clock. These signals enable the bus traffic to be classified in accordance with internal processor activity. If they are not available, it is very difficult to separate out instruction fetches from other cycles. Personality modules used in logic-state analysers use complex state-machines to separate memory cycles into their various categories. The other flags relate to the internal state of the processor, and thus they are useful for object decomposition at the system level.

The software probes used to update the monitor register in the hardware probe (section 6.5.2) interfere with the operation of the software. Interference can be reduced by the inclusion of monitoring features in the microprocessor. Instead of handling the module-identifier stack in software, a hardware push down stack can be included.

Monitor-register addresses are permanently allocated: one to each stimulus monitor-register, and one to the module-identifier monitor-register, which is also the top of the push down stack (figure 8.1). Probe instruction (figure 8.2) are added to the assembler instruction set. Opcode modifiers determine which of the probe algorithms is executed by the probe

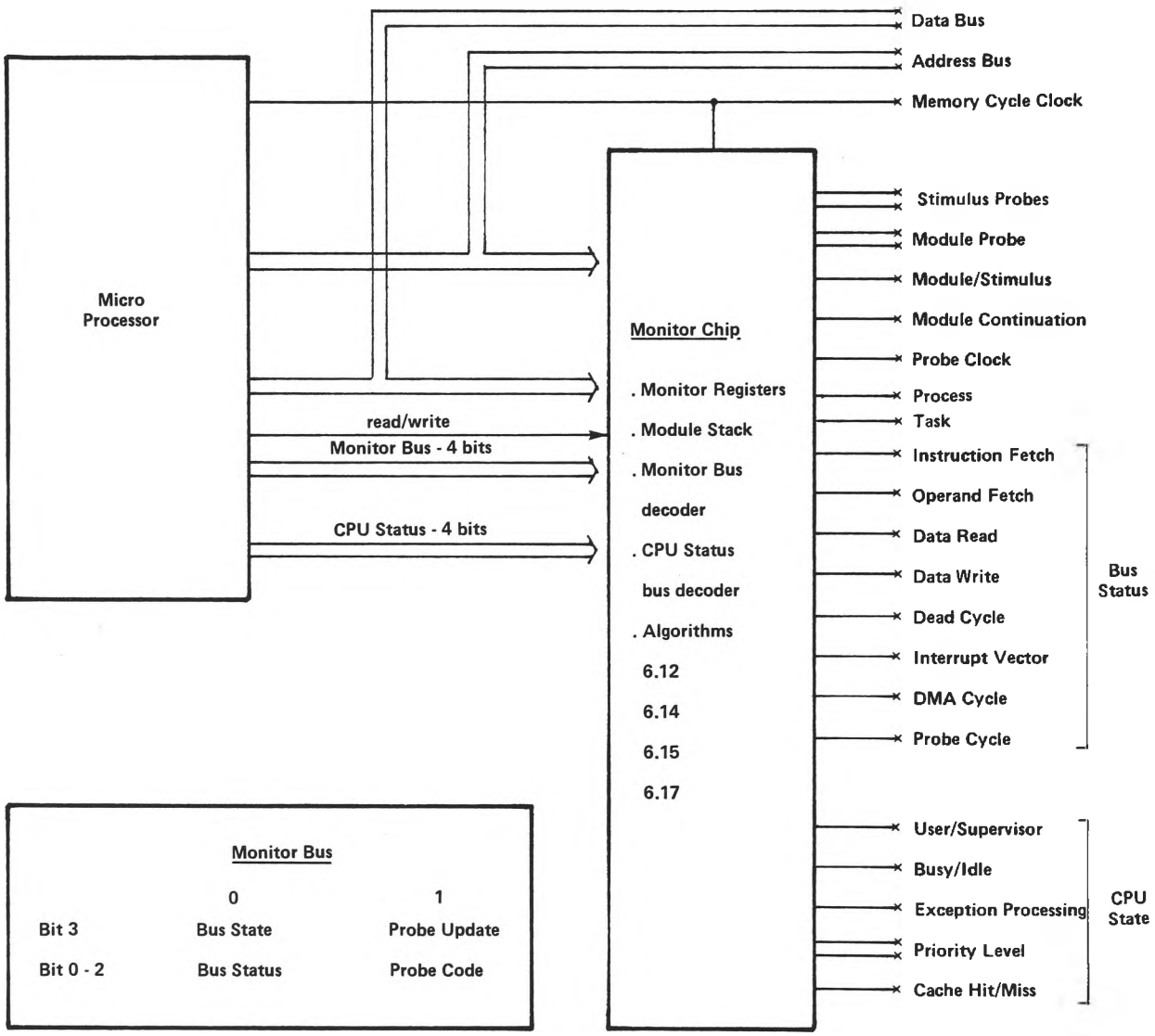


Figure 8.3 Microprocessor to Monitor Chip Interface

hardware. The instruction operand has all the addressing modes available on the microprocessor, although intermediate (operand is value) and direct (operand is address of value) will be most commonly used. In these addressing modes, only one instruction is needed to implement probes, considerably reducing probe interference. Addressing modes involving registers will create more interference because the register has to be loaded with the address of the probe value.

The probe instruction is able to execute in one instruction cycle because:

- the probe addresses are fixed and do not have to be included in the instruction,
- the operand of the instruction is either the value to be stored into the monitor register or the address of the value,
- stack handling, probe clock generation, and status flag settings are all implemented in hardware as selected by ~~to~~ the opcode modifiers, and
- the processor latches the data directly into the monitor registers.

Implementing the additional hardware inside the microprocessor chip is no problem, but getting the signals out is. The probe requires at least 40 pins, assuming 16 bit monitor registers, which cannot be found on an existing microprocessor. Also, the defined hardware will not handle pre-emption, which requires separate stack areas for each process, considerably complicating the hardware. Therefore, the probe hardware has to be implemented on a separate monitor chip (figure 8.3).

Two additional busses connect the microprocessor to the monitor chip: a four bit monitor bus, and a four bit status bus. Some of the 16 and 32 bit microprocessors already have a status bus (for example the function codes on the M68000). In this way, the pins required for monitoring are reduced to eight. The status bus contains information about the current status of the processor: busy or idle, user or supervisor mode, etc. This information can be used for system level measurements.

During normal operation, the monitor bus carries information about the current bus cycle:

whether the cycle is an instruction fetch, data write, etc. This bus is decoded in the monitor chip to produce the memory-cycle-clock qualifiers. When a probe instruction is executed, the monitor bus contains the probe opcode modifier, enabling circuits in the monitor chip to execute the desired probe algorithm. The data on the data bus, which is fetched by the probe instruction, is latched into the appropriate monitor register at the end of the instruction cycle. In the case of the probe on a return from interrupt (figure 8.2), a continuation flag is automatically set; in all other probes it is reset. Thus, module fragmentation can be detected by monitoring the continuation signal.

Pre-emption can be handled by allocating a separate stack frame for each process. When a process-stimulus-probe instruction (figure 8.2) is executed, the top-of-stack pointer must be relocated to the stack frame for that process, if a stack frame exists, or a new stack frame allocated and the top-of-stack pointer moved there. Research into the nesting depth of high-level languages (Patterson and Sequin 1982) indicates that the nesting depth is greater than eight for less than one percent of calls. Thus, allocating a stack frame greater than eight (for example 16) will handle the majority of nesting situations. If a stack overflow does occur, a special event-marker can be inserted in the event trace (or a control flag set) to indicate this. This condition will be of considerable interest to the analyst trying to understand the program. By incrementing a counter each time a stack overflow occurs on module entry, and then decrementing it on module exit, the chip should be able to maintain probe balance, and correctly instrument the process, except for the loss of those probes below the maximum stack depth.

The outputs of the monitor chip will be connected to probe connectors, in parallel with the address bus, the data bus, and the memory-cycle clock. The probe clock will be synchronised to the next instruction fetch, and as the fetched instruction is from the instrumented program, the current program counter can be read in parallel with the monitor registers. By reading the address bus and data bus during a probe cycle, the placement of probes; both address of probe instruction and probe data, or address of probe data; can be checked.

A computer system built from these components could be instrumented permanently with

very little interference to the software. In fact, removal of accounting programs from the computer to the external tool would more than compensate for probe interference. In addition, the outputs of the monitor chip could be used for hardware fault diagnosis.

In the design of the RISC (Patterson and Sequin 1982), a number of constraints were placed upon the architecture so that the goals of simplicity and effective single chip implementation could be met. These constraints were:

- execute one instruction per cycle,
- make all instructions the same size,
- access memory with only load and store instructions, and
- support high-level languages.

The first constraint eliminates variations in the execution-time of a path due to variations in instruction execution time. Thus, removing variations in the execution time of machine code is not only desirable from the measurement point of view, but also simplifies the design of the processor.

Other causes of path execution-time variation can be detected and compensated for by attaching counters to the Bus State signals out of the monitor chip. The cache hit/miss signal would be supplied by the cache circuitry, if a cache is used.

8.4. Instrumentation and Measurement of a Small Computer System

A small computer system, used to multiplex terminals over a Cambridge-Ring local-area-network to a host Unix system, was designed with performance measurement as one of the design goals. The system has been instrumented in accordance with the methodology discussed in chapter 6. The hardware, an M6809 system with DMA capability, was designed by Meng Fong, and constructed by Michael Milway. The system, and applications, software was implemented by Gary Stafford. The hybrid monitoring-tool was used to measure system performance, monitor program execution, to find areas of poor performance, and to identify the cause of hardware, and software, bugs.

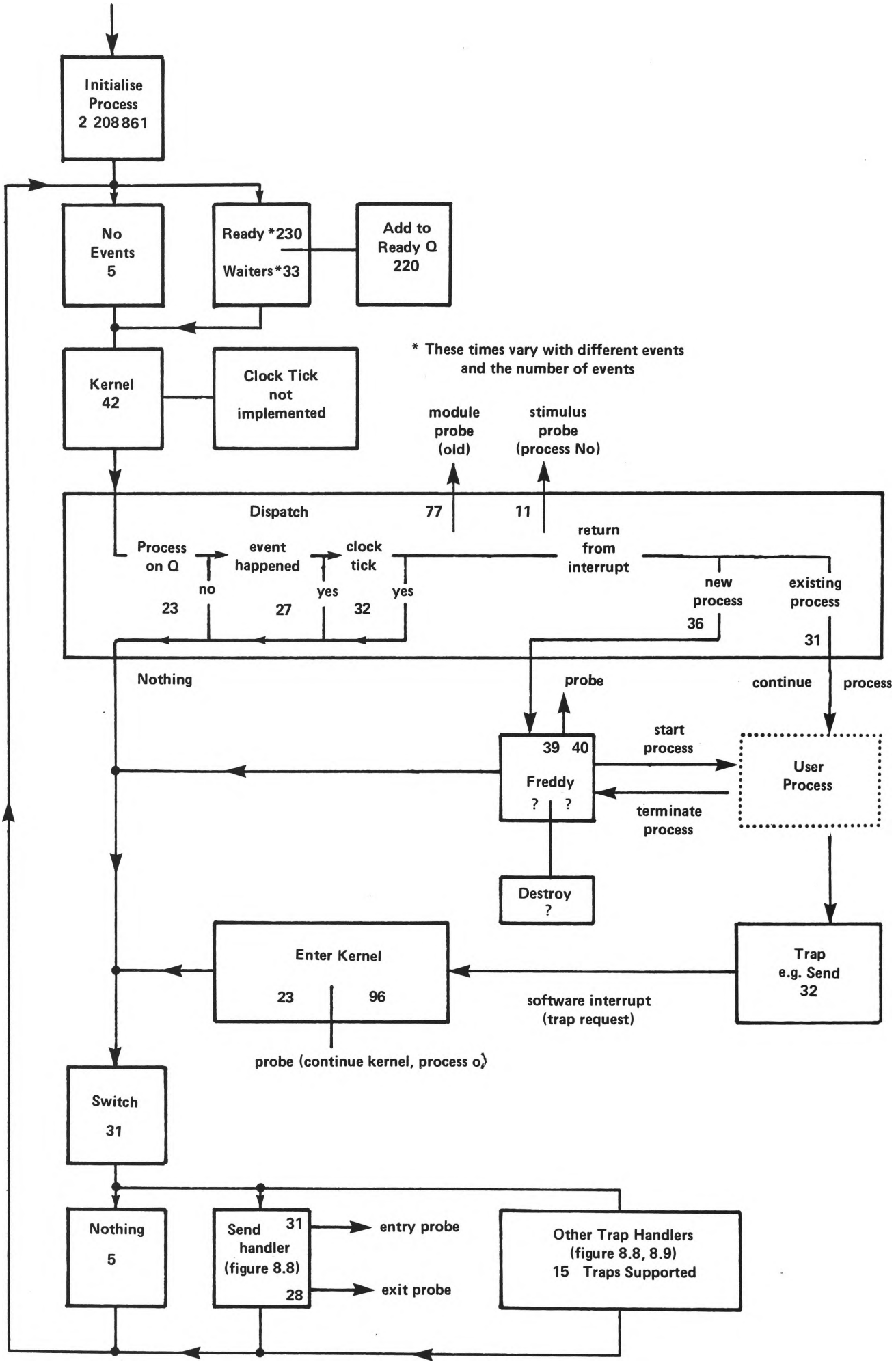


Figure 8.4 Execution-Path Flow-Graph of the Kernel of a Message-Passing Operating System (after modification to Ready_Waiters) - all times in microseconds.

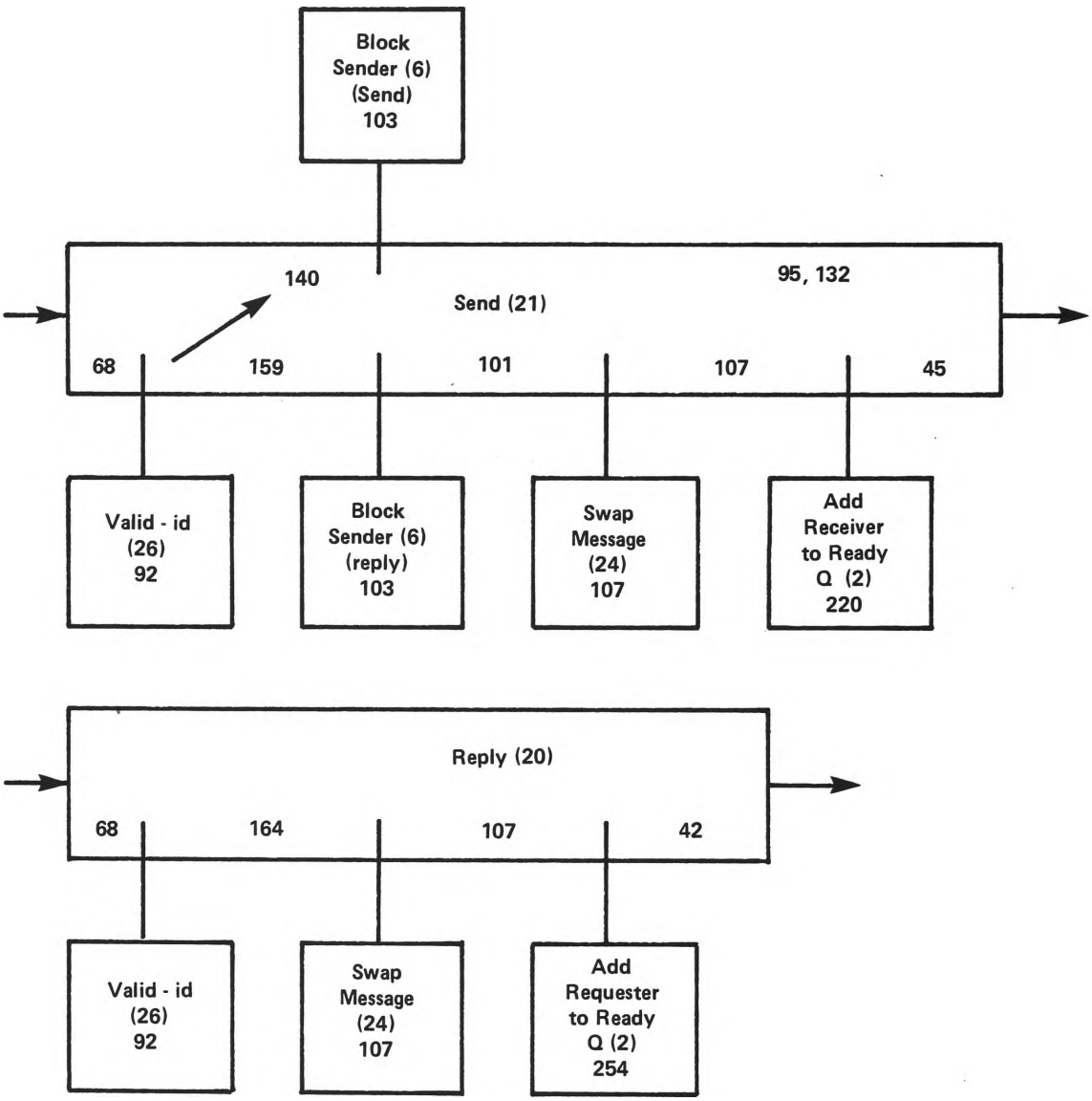


Figure 8.5 Execution-Path Flow-Graph for the Send_Message and Reply Trap-Handlers (before modifications to Valid_id, Block, and Swap_Message) - all times in microseconds - numbers in brackets are module numbers.

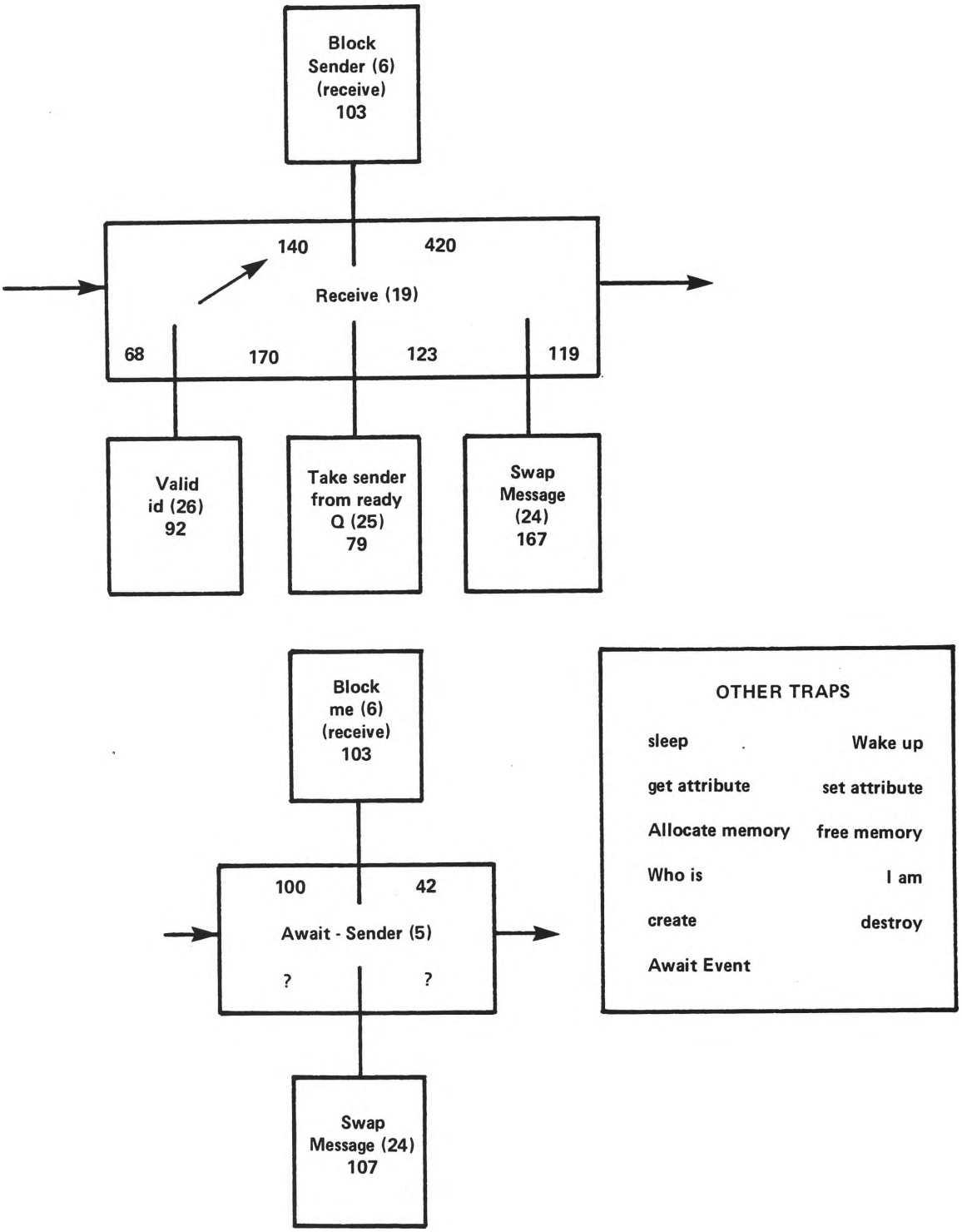


Figure 8.6 Execution-Path Flow-Graph for the Receive_Message and Await_Sender Trap-Handlers (before modifications to Valid_id, Block, and Swap_Message) - all times in microseconds - numbers in brackets are module numbers.

The operating system is a single address-space **message-passing system** similar to Thoth (Cheriton et al 1979). Sixteen primitive requests are supported by the kernel (figure 8.4). Four of these (figures 8.5 and 8.6) are message-passing primitives. When a process requests a message to be sent, the message is passed to the receiver by exchanging pointers to message fields, the sending process is blocked waiting for a reply, and the receiving process is added to the ready queue. When the receiving process replies, the pointers are swapped again, and the sending process is added to the ready queue. The receive and await-sender primitives enable a process to block waiting on either a message from anyone or a message from a specific process. The other primitives handle requests like: create a process, allocate memory (figure 8.6).

When a process requests a service from the **kernel**, a software interrupt vectors to the enter-kernel procedure (figure 8.4), which handles the process switch. Then, the appropriate primitive is executed to perform the requested service. After which, a check is made of the event vector to see if any events have happened, and if so, the process waiting for that event is added to the ready queue. Then, the dispatch-process procedure switches to the first user process on the ready queue. This may not be the process which requested the service, for example, when the request is a send message the sending process is removed off the ready queue so that it can wait for a reply.

This system consists of a large number of small procedures (30 in the kernel, 27 for terminal handling,etc) which are executed in sequence to form a process (figures 2.4 and 8.10). A task consists of a sequence of processes (figure 2.3) to perform some operation. The system consists of a number of terminal handling tasks (2 for each terminal), and three network handling tasks (input, output, protocol). Many of these tasks use the same processes (which use the same modules), and thus, instrumentation at the task level must be sufficient to enable the interleaved tasks to be separated out.

In the discussion of instrumentation that follows, the procedures are at the module level (block-structure level), the processes are considered to be at the program-execution level, and the tasks at the task level. The names chosen for the levels in the object hierarchy (figure 2.1)

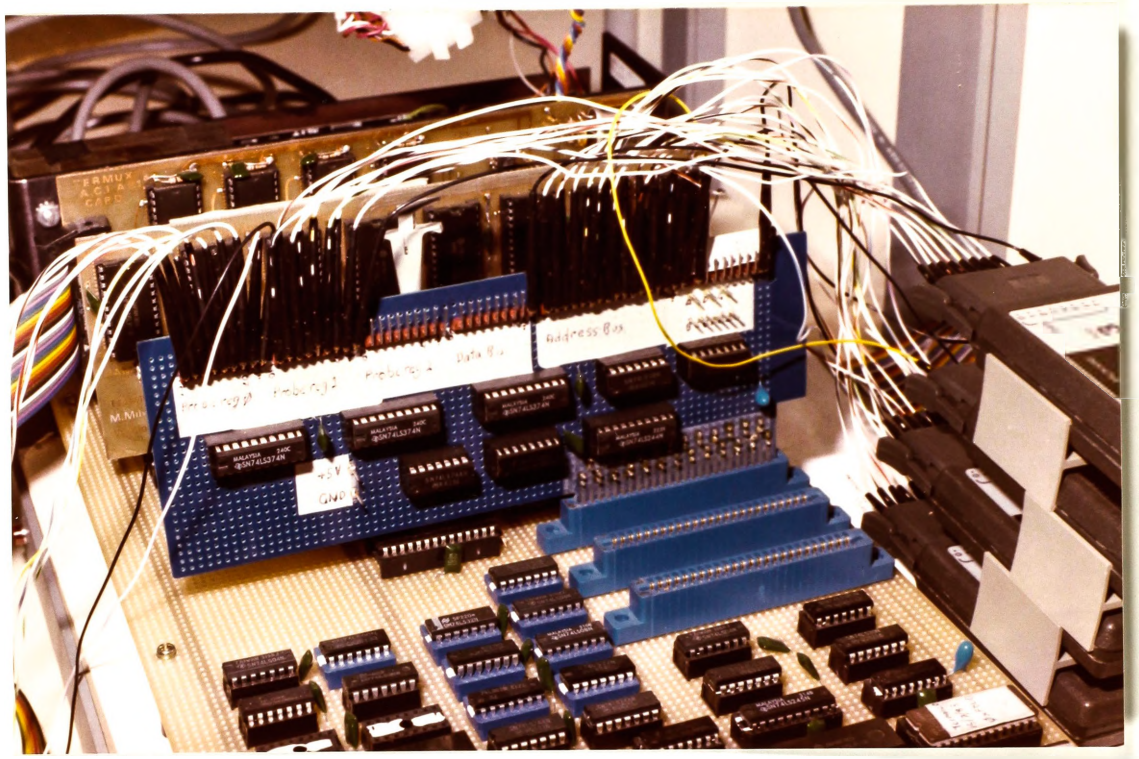


Figure 8.7 Hardware Probe Plugged into Experimental Terminal Multiplexer

are ~~ment~~ meant to map into the terminology used in the system under discussion. Here we will talk about machine-code level, module level, process level, task level, and system level, keeping in mind the above mappings.

In the following sections, the method of instrumentation is discussed first, followed by a description of the uses to which the instrumentation has been put.

8.4.1. Instrumentation

A **hardware probe** (figure 8.7), with the same basic circuit design as the apple probe (figure 7.3), was plugged into an input-output slot on the terminal multiplexer, allowing the hybrid monitoring-tool (figure 6.12) to be used. The tool was used in two modes: as a logic-state analyser for fault finding, and as a hybrid monitor for performance measurement and evaluation.

A minor design error was found in the probe during these measurement experiments. As designed, the circuit will produce a clock pulse (figure 7.2) every time a value is stored in a monitor register. However, if two monitor registers are written to on successive memory cycles (for example a 16 bit store over an 8 bit bus) only one clock pulse occurs not two. This reduces the number of events in the event stream, without data loss, but if the measurement tool is expecting separate module and stimulus clocks it may be confused. One solution is to AND the probe clock with the memory-cycle clock. Other solutions are to have a separate clock for each monitor register, or a single probe clock with the register number as a clock qualifier.

As the operating system is pre-emptive, the process number of the executing program is an important piece of stimulus information. One monitor register is used to store process numbers, and a clock which indicates the updating of this register was obtained, enabling process-level measurements (figure 2.3). Thus, the ability to obtain a separate clock for each monitor register is important, and the hardware probe (figure 7.3) should be modified to provide either separate clocks or clock qualifiers, depending on the clock qualification abilities of the logic-state analyser.

All procedures in the system, except some assembler routines, were instrumented with

```
static __Freddy()                    Example of an Instrumented Program
  {
   Entry( 26 );                      Call _____en routine to write module no 26
   (*_____uo)( Who__sent(), Active__pd->msg );
   Destroy( Active__pd->id );
   Exit();                           Call _____re routine to write old module no
  }

*       line 500, file "KERNEL/Basics.c"   Entry(26) - Compiler Output
        ldb     #26
        stb     0               write to probe - to here 7 microseconds
        ldd     #26              3 microseconds
        pshs    d                10 microseconds
        jbs     _____en+0        call entry routine - 19 microseconds
        leas    2,s              5 microseconds
*       line 501, file "KERNEL/Basics.c"
        "       other assembler instructions
        "
*       line 503, file "KERNEL/Basics.c"
        "
        "
*       line 504, file "KERNEL/Basics.c"   Exit()
        jbs     _____re+0        call exit routine - 19 microseconds
        tfr     u,s
        puls    x,y,u
        rts                      return to routine which called __Freddy

            .globl _____en       Code of probe used at routine entry
            .text
_____en:    pshs   x             execution time 93 microseconds
            ldd    modstack
            ldx    modindex
            leax   1,x
            stx    modindex
            leax   d,x
            ldd    4,s
            stb    ,x
            tfr    d,x
            leax   d,x
            ldd    #1
            addd   __Mod__ref,x
            std    __Mod__ref,x
            puls   x,pc          return to __Freddy

            .globl _____re       Code of probe used at routine exit
            .text
_____re:    pshs   x             Execution time 60 microseconds
            ldd    modstack
            ldx    modindex
            leax   -1,x
            stx    modindex
            lda    d,x
            ora    #x'80
            sta    0             write to probe - instruction time 5
            puls   x,pc          instruction time 11 microseconds
```

Figure 8.8 Software Probes Added to Experimental Terminal Multiplexer - A seperate stack is used as a probe stack - all times in microseconds.

Example of an Instrumented Program

```
main()
 {
     _____Entry(5);        macro to write module number on entry
     _____Dropout()        macro to write old module number during exit
 }
```

Compiler Output

```
LL0:
        .data
        .text
        .globl    __main
__main:
        pshs    x,y,u       function call entry code
        tfr     s,u
        leas    F1,s

*       line 3, file "x.c"
_____Entry(5);              Code of probe macro used at routine entry
        ldb     _____T1____+0   execution time 23 microseconds
        stb     -1,u
        ldb     #5
        stb     0           write to probe - to here 16 microseconds
        ldb     #133
        stb     _____T1____+0

*       line 4, file "x.c"
_____Dropout();             Code of probe macro used at routine exit
        ldb     -1,u        execution time 14 microseconds
        stb     _____T1____+0
        stb     0           write to probe
        tfr     u,s         code for return from function call
        puls    x,y,u
        rts
```

Figure 8.9 Software Probes Modified to Use the Process, and Procedure, Stacks and to be called as Macros not as Subroutines.

Implementation		Time in microseconds		
		Before	After	Total
Entry	seperate stack	7	125	132
Exit	seperate stack	68	11	79
Entry	process stack	16	7	23
Exit	process stack	14	0	14

Table 8.1 Cost of Software Probe - on the Experimental Terminal Multiplexer - Probe code given in Figures 8.5 and 8.6.

software probes indicating module entry and module exit, using algorithms 6.14a and 6.15a. These probes write module numbers to the first of the three monitor registers. Stimulus probes in the dispatch and enter-kernel modules (figure 8.4) write process numbers to the second of the three monitor registers.

Module-number probes were implemented in two ways: the first (figure 8.8) used a separate stack as a module-identifier stack, and the second (figure 8.9) used the first local-variable in each procedures stack-frame. The execution time (table 8.1) of the second method is considerably shorter than that of the first method for two reasons. First, the probes in figure 8.8 are called as procedures, thus, adding 30 microseconds to the execution time (most of this time is spent saving and restoring registers during the call and return). The probes in figure 8.9 are written as macros, and thus, are inserted into the code, consequently, execution time is saved at the cost of increased memory usage. Secondly, the probes in figure 8.8 spend time finding the stack pointer for the module stack (including stack frame); all of which is eliminated in the probes in figure 8.9 because the current-procedure's stack-frame is used.

The module entry and exit probes (figure 8.9) are based on algorithms 6.14a and 6.15a, and T1 is a global temporary-storage variable. The entry and exit kernel module-identifier probes (located in enter-kernel and dispatch - figure 8.4) were modified to accommodate the change in module sequence caused by pre-emption. On entry to the kernel, the module identifier of the pausing module is placed in a field in the process descriptor of the pausing process. When a process is dispatched, the module identifier for the resumed module is obtained from a field in the process descriptor of the dispatched process. This is done to guarantee that the correct module identifier is written when a process is resumed.

With the alternate algorithms, the module identifier of the currently executing module is held in the global variable, and thus, at a process switch it must be saved or the module probes in the next process will overwrite it. With algorithms 6.14 and 6.15 the module identifier of the currently executing module is already saved on the stack when pre-emption occurs. However, with these algorithms a more complex, and hence more expensive, stacking procedure is needed,

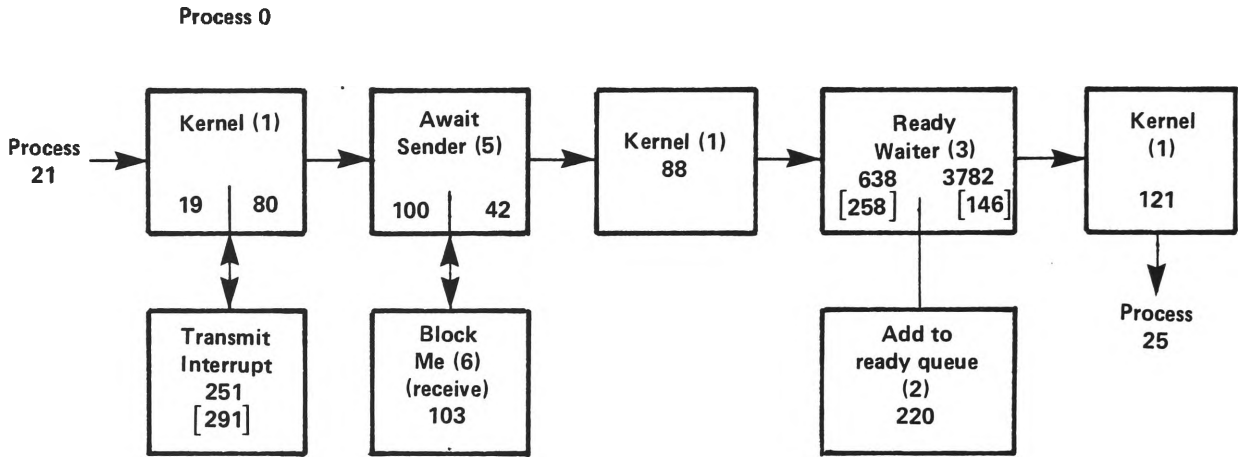


Figure 8.10 Execution Path of the Kernel Process during a transmit interrupt (figure 2.3) - numbers in square brackets are execution times after modifications to Ready_Waiters - all times in microseconds.

because we need access to the previous module number as well as to the current module number. An area in the current process's stack-frame should be used, not a single field in the current procedure's stack-frame.

One problem with using macros is that the "C" compiler has facilities for the inclusion of assembler macros but the assembler doesn't. Consequently, the insertion of instrumentation into the assembler procedures had to be done manually. As a result, a number of the assembler procedures were not instrumented. Measurements of these procedures had to be made at the **machine-code level** using the system's address map to find entry and exit points. Care has to be taken when combining measurements made at different levels in the object hierarchy to correctly align the measurements. Measurements made at the machine-code level, with a purely hardware tool, are from routine entry to routine exit; where measurements made at a higher level, using a hybrid tool, are from probe to probe.

Combined machine-code-level measurements and module-level measurements were used to measure the execution path of the kernel process (figure 8.10). The execution-path flow-graph of the terminal administrator process (figure 2.4) was measured at the module level only. The cost of instrumenting a system at the module level is reasonably high (table 8.2), and in the final system the module-identifier probes will be removed, leaving only the process-number stimulus-probes.

Stimulus probes were placed in the dispatch and enter-kernel routines (figure 8.4). These probes write the process number of the process being dispatched, or the kernel, to the second monitor register. These probes were used to measure the sequence of processes in the character-handling task (figure 2.3), in conjunction with machine-code-level measurement of the interrupt handler, and module-level measurement of the first kernel process. To completely instrument the system at the **process level** the following stimulus probes are needed:

- process number in dispatch and kernel process-number in enter-kernel,

Probes	Execution Time	
	microseconds	% of task
Block-Structure Level only (module probes)	1340	10.6
Program-Execution Level only (process probes)	198	1.7
Task Level only	264	2.2
System Level only	264	2.2
Total Instrumentation (understanding)	1703	12.8

Table 8.2 Cost of Instrumenting the Character-Handling Task at various levels in the object hierarchy.

- idle (nothing in figure 8.4), and
- one event indicator in each interrupt handler.

All probes at lower levels in the object hierarchy can be removed. Thus, the cost of instrumenting this system at the process level is low, for example 198 microseconds in the case of the character-handling task (table 8.2). Each stimulus probe takes 11 microseconds. Notice that no probes have to be inserted into a user process in order to monitor it at the process level (figure 8.4).

To instrument the system at the **task level**, some additional stimulus probes are needed to indicate which interrupt occured (one handler can handle several interrupts), and in the case of a program like the terminal-administrator, which handles all the terminals, which terminal it is handling. This additional stimulus information indicates which task is being performed by each process. The measurements recorded here are for a single task, and thus, this extra stimulus information was not needed.

Measurement of a single task was sufficient to optimise the kernel (section 8.4.2). Probing at the task level costs little (table 8.2), but it provides all the information needed for **system-level measurements**, and for **cpu-usage accounting** by task, as well as system-level measurements. By adding customer information, and disk usage data, to this task-level event-trace data, the system manager has enough information for customer billing. Additional stimulus probes (for example queue lengths) can be added to provide engineering data for system modelling, and for performance studies.

In order to **understand** the operation of the system, all the above probes, plus some additional stimulus probes, are combined to produce a totally instrumented system. Thus, we have:

- module entry and exit probes in every procedure,
- process number probes in the enter-kernel, dispatch, and idle (nothing) procedures,
- stimulus probes in the interrupt handlers, and in some processes, to indicate which task is being executed, and

- appropriate stimulus probes in various modules to explain why the the system is doing what it is doing.

The selection of **appropriate stimulus information**, for the purpose of understanding the operation of a system, has not been dealt with in a practical way in the previous chapters, except to state that it is dependent on the function of the object under study. The experiments we have conducted on this system have revealed which stimulus data is usefull for understanding the operation of the kernel, and the terminal handling processes. Much of this information can be gained intuitively by studying the execution paths, but instrumentation gives a greater guarantee of correct answers.

The purpose of stimulus probes in understanding is to answer the questions: who requested this operation, why was it requested, and which operation, of the ones which the object can perform, is being performed. We have already seen the use of stimulus probes for answering the question: who requested this operation, when we looked at the identification of processes and tasks. Most of the time spent in the kernel is spent executing primitives to service user requests. A stimulus probe can be added to the kernel to record which primitive has been requested. In the send primitive, the process number of the receiving process is an appropriate piece of stimulus information. A number of primitives call the block procedure. Appropriate stimulus information for the block procedure is which process is to be blocked, and why is it to be blocked (e.g await reply).

As the interrupt handlers are re-enterant, an appropriate piece of stimulus information is which event is being serviced (e.g which terminal, and was it receive or transmit). Interrupt handlers which handle several events of the same type may do so during one invocation of the handler, thus the stimulus probe may occur several times. An appropriate piece of stimulus information for the terminal administration process (figure 2.4) is the character being handled, as different characters cause different execution paths. This stimulus probe could be placed in the cooked procedure. During the design, and development, of a system, the variables which aid in understanding the operation of the system become fairly obvious, particularly if monitor-

Probe	Execution Time in microseconds	% of Task Time		
		original	final	savings
Module + Stimulus before modifications	1798	8.18		18.35
Module + Stimulus after modifications	1354		10.46	13.8
Stimulus only	198	0.95	1.70	2.00

Table 8.3 Cost of Software Probes Inserted into the Character-Handling Task (figures 2.3,2.4,8.4). See Errata point 2.

ing tools are being used to debug and measure the software as it is developed.

All the measurements recorded in this chapter, and in chapter 2, were made on this system with these tools. The times in these figures include probe times, because the probe time is short; but the times in the measurements recorded in chapter 7 do not include probe times, because of the high cost of the probes on that system. The **cost** of the instrumentation inserted into the character-handling task (figure 2.2) is recorded in table 8.3. The cost of instrumenting this task at various levels in the object hierarchy can be found in table 8.2.

During these measurement experiments, a number of **limitations** were found with the hybrid performance-monitor. The object-generation algorithm (A6.4) simply states: find the next object and record the modules in it. The implementation of this algorithm (figure 6.13) searches for a start module-number and an end module-number to delimit the object. This implementation was modified (section 13.1) to allow for minor variations in path execution-time, due to limited measurement resolution, and to enable the measurement of a single module, by specifying identical start and end module-identifiers. Measurement of a single module is the easiest way to obtain a module spectrum (figure 2.5 was done this way).

The measurement of a single module would have been enhanced even further if the logic-state analyser could be set up to record a specific module-identifier and the next record (or when the module has stimulus probes, two or three records) in the event trace, without having to specify what that event is. Often modules of interest for spectral analysis perform a variety of functions, and consequently, are followed by one of a number of different modules, depending on why the module has been called. As the logic-state analyser doesn't have this feature, the complete event trace had to be recorded and searched for invocations of the module. Also the depth of this analyser (64 words) is far too small, and continuous recording (circular list or double buffering) is desirable.

The above discussion illustrates the simplistic nature of the object detection software in the hybrid performance-monitor. An additional problem which highlighted the limitations is the fact that many processes are in infinite loops. For example, the input handling process (figure

Task	Execution Time in microseconds	Savings	
		Time	% of original
Origional	21993		
After Ready_Waiters Modified	14347	7646	34.8
After Swap_Message, Block, and Valid_id Modified	13027	1320	6.0
After Cooked Modified	12944	831	3.8
Total Savings		9797	44.6

Table 8.4 Execution Times of the Character-Handling Task (figure 2.3) before and after modifications.

2.3) is normally blocked waiting for an event to occur. When the event occurs, it sends a message to the terminal administrator, waits for a reply, and, after it gets the reply, waits for an event again. Consequently, the module probes for this process always indicate continuation, and thus, we are unable to define unique object start and end points in terms of module numbers. The way to delimit the object, in this case, is to record all modules while a particular process number is sitting in the stimulus probe, plus the module before. The concept is: measure at one level while the probe for a higher level in the object hierarchy has a value which identifies the object of interest. This is more satisfactory than counting the number of module invocations, because this number may not be a constant, and it conforms to the hierarchical basis of the performance measurement formulation.

8.4.2. Performance Measurement

One obvious outcome of these experiments is the data recorded in the figures in this chapter and in chapter 2. Studying this data pinpointed processes with poor performance. Studying the process execution-paths, and the module execution-times, pinpointed modules with poor performance. The modules with poor performance were not the ones that the programmer would have chosen to optimise. In fact the greatest saving was made in a module that he would never have considered.

A number of **performance improvements** were made to the character-handling task (figure 2.3). As most of the improvements were made in kernel modules, the kernel was optimised, and hence, the performance of all tasks was improved. After each modification, the task was measured again to ascertain the performance improvement, to see if any bugs had been introduced (one was), and to verify that the programmer had done what he said he had done (he had forgotten about modifying one procedure). The execution time of the character-handling task (table 8.4) was almost 22 milliseconds, including probe costs. From the event-trace graph of this task (figure 2.3) it can be seen that the process taking the longest time is the kernel immediately after an interrupt. When the these kernel processes were looked at (figures 6.14 and 8.10), the module consuming the largest amount of execution time was found to be ready waiters.

When an interrupt occurs, a flag is set in an event-happened vector, which is 64 flags long. Ready waiters searches the vector looking for events which have happened, and when it finds one, adds the process waiting for that event to the ready queue. The code consisted of a for loop, a test, and a call if the test returned true. All the time was being used testing for non-existent events. The loop was modified to search between minimum and maximum values, saving 4.016 milliseconds on a single transmitX event (figure 8.10). The interrupt handlers had to be modified to set the minimum and maximum event variables, adding an extra 40 microseconds to the execution time of the transmitX interrupt-handler. The total saving (table 8.4) for the character-handling task was 7.646 milliseconds, or 34.8% of the original execution time. As this routine is in the kernel, and is executed in response to every interrupt, this saving represents a considerable optimisation of the kernel. The process execution times after this modification are shown in figure 2.3 in square brackets.

The next largest execution time of a kernel process in the character-handling task was the time taken to send messages. The swap-message, block, and valid-id procedures were rewritten in assembler, reducing the execution time of the send-message kernel-process by 330 microseconds (curly brackets in figure 2.3 show the reduced execution times). One hundred and eleven microseconds were saved by the removal of the module probes from these processes, and some saving was due to the function calls being sped up, because there is no need to save all the registers on the stack (which "C" does). The add-to-ready-queue procedure (figure 8.5) was not modified because it is complex, and thus, difficult to implement correctly in assembler. This modification affected other primitives also, resulting in a total saving of 1.32 milliseconds, or 6 percent of the execution time of the character-handling task.

The largest user-process execution-time was that of the terminal administrator (figure 2.4). The execution-path profile for this process (figure 2.6) indicated that the most frequently used path was the handling of simple characters. This path is one of the fastest (figure 2.8) paths through the process, but it had the highest utilization (figure 2.10). The modules with the longest execution-time (figure 2.9) are cooked, delete, and tab. The only one of these modules used

in handling simple characters is cooked. Investigating the execution-times for various paths through cooked (figure 2.4) revealed that handling a simple character took longer than handling either a backspace or a return character. A few lines of code were added to cooked to give priority to handling simple characters, resulting in a saving of a further 3.8% of execution time (angle brackets in figure 2.3 give the new execution times).

While studying this task, we noticed that the transmitt interrupt appeared to occur in the enter-kernel procedure as soon as interrupts were re-enabled. This observation prompted the question: were the interrupts enabled when they should be? Investigation of the code raised the level of suspicion even further, but measurement proved inconclusive. The time taken between when the character was sent and the transmitt-complete interrupt was 1161 microseconds. At 9600 baud, a character is transmitted in 1047.6 microseconds. This time delay resulted in the transmitt interrupt occuring during the enter-kernel procedure when interrupts were disabled. A different experiment had to be devised to answer the question.

The **total saving from these modifications was 44.6% of the original execution-time** of the character-handling task (table 8.4). At this stage, we decided that no further optimization should be done to either the kernel or the character-handling task. One problem for the analyst is that the availability of accurate measurements makes the task of optimisation so easy that system changes obsoleted the measurements rapidly.

The **cost of the software probes was 8.18% of the original execution-time**(table 8.3), but more significantly **18.35% of the savings**. Thus, the inclusion of measurement facilities at design time was paid for many times over by the resultant savings. Once system optimisation is complete, the module identifier probes will be removed, leaving only the process-number stimulus-probes, reducing the artifact to 1.7% of the execution time. The process probes are being left to enable easy study of system operation if problems arise.

The hybrid monitoring-tool was connected to the system during the development period, and was regularly used for **fault finding**. When the system hung, an event trace at module level was recorded. This indicated which modules, if any, were being executed. In the case of a tight

loop within a module, an event trace at machine-code level was recorded and the area of code pinpointed. The probe registers are at physical addresses 0, 1, and 2. A number of invalid pointers (value 0) were found from observations of rubbish in the module-identifier event-stream, due to spurious writes to location zero. At system start up, all memory is cleared to zero, increasing the chance of invalid pointers being zero. When a memory location (code or data) was being corrupted, all writes to that location were monitored to determine which code module was causing the corruption. The time saved in finding software bugs, because instrumentation was available, is significant, but it became second nature to use the tool, making estimation of the time saved impossible.

In addition to finding the cause of a number of software faults, the tool was used to investigate faults which could have been either software or hardware. Traditionally, this is the most difficult area of fault finding. The hybrid monitoring-tool was used to determine what the software was doing when the fault occured, and then, if it appeared to be a hardware problem, hardware signals of interest were connected to the logic-state analyser section of the tool so that hardware operations could be monitored.

One such fault was that the priority levels in the priority-interrupt controller were not being set up correctly, because of data loss caused by a five nanosecond delay in a timing signal. A second, similar fault occured because the input-output address space was not completely decoded, and consequently, the interrupt controller was selected by addresses at two different locations. The initialisation routine set up the controller at one address, and then, as part of a zero memory phase, cleared the contents out at the second address.

In **conclusion**, instrumentation of a system at design time has proved its worth in performance optimisation, has proved invaluable in finding hardware and software faults, saved time both in fault finding and optimisation, and led to an increased understanding of the system. These experiments have contributed to the validation of the performance measurement formulation, and have shown the practicality of a performance measurement methodology based upon that formulation.

9. Measurement of Multi-Processor Computers

Measurement of the performance of multiple processor systems is a relatively new and difficult field. Most of the work to date has been done by research organisations who have built their own processor systems. Array processors are now available commercially, with image processing being a major application, heightening the need for performance measurement.

Parkinson and Lidell (1983) define the major questions asked by potential users of multiple processor systems:

- What class of problems are highly suitable for a given multiple processor system?
- What class of problems are highly unsuitable for a given multiple processor system?
- What type of performance is it reasonable to expect from a given multiple processor system?

These questions are very difficult to answer in an unambiguous fashion.

A significant contribution to the complexity of the measurement problem is the multitude of ways in which processors can be interconnected. Multi-processor architectures range from highly parallel structures, where processing elements are tightly coupled and used to solve only one problem at a time, to loosely coupled systems where each processor is working on a different problem. Highly parallel machines (Haynes et al 1982) can be divided into the following categories: multiple special-purpose functional units, associative processors, array processors, data flow processors, functional programming language processors, and multiple cpu's. Data communications is seen to be the key to the successful exploitation of parallelism, but there are a number of ways of interconnecting processors (cross-point switches, ring networks, systolic arrays, banyan networks, cube networks, tree structures), each with its own advantages and problems.

In addition to the variety of processing methods and the variety of ways of interconnect-

ing processing elements, there is a variety of ways in which the resultant machines can be used. Illiac IV (Barnes 1968), the pioneer of parallel processing, processed many data sets under the control of a single program, using many identical processing elements. At the other end of the spectrum, we see parallel execution of independent jobs using either monoprogrammed processors or multiprogrammed processors. In the middle, we find different parts of the one job distributed among several processors, each passing data to the other.

Many performance evaluation studies of multi-processor computers have been concerned with measurements at a high level, for example measurement of the execution time of algorithms, partly because of the difficulty of measurement and partly because of the newness of the field. Questions of interest to researchers include:

- What is the best way to configure the processing elements to solve a particular problem?
- How much overhead is generated by managing parallelism?
- What effect does varying the number of processing elements have on performance?
- What effect does changing the communication path configuration have on performance?
- How do you control and measure the synchronisation of parallel processes?
- How do you instrument a parallel system for performance evaluation?

In the sections that follow, performance measures, measurement tools and measurement techniques reported in a number of papers are discussed. The measures used are compared to the measures defined in extensions of the formulation of performance measurement developed in chapter 2 for single processor systems.

9.1. Performance Measurement of SIMD Machines

Siegel et al (1982) have proposed a set of nine measures, based upon the work of Kuck (1977), for evaluating the performance of algorithms on SIMD (single instruction stream-multiple data stream processor) machines. A SIMD machine typically consist of a control unit, a set of N

processing elements (each with its own memory), and an interconnection network. The control unit broadcasts instructions to all processing elements, and each active processing element executes each of these instructions on the data in its own memory. Each instruction is executed simultaneously in all processing elements. The interconnection network allows data to be transferred among processing elements. The processor can be configured so that all processors work upon the one data set, or so that each processor works upon separate data sets.

In applications where the same operation is repeated thousands of times, or where computationally intensive matrix and vector operations are frequently used, SIMD machines promise to reduce the computation time. The complexity of SIMD algorithms is a function of the size of the data set, the number of processing elements used, and the structure of the interconnection network. Obvious uses of performance measures are: to select between alternative algorithms, to study the effect of varying the number of processing elements on the performance of an algorithm, and to study the effect of varying the size of the data set on the performance of an algorithm.

9.1.1. SIMD Object Hierarchy

Before discussing Siegel et al's set of performance measures, the formulation of measurement is extended to cover SIMD machines. Extensions are required to the formulation to take into account the parallel operation of several processors. The conceptual model of an object during execution is the same: a sequence of modules. However, the hierarchical decomposition of a SIMD machine is slightly different to the decomposition of a monoprocessor.

The highest three levels: System, Task, and Program Execution, are the same. Below this a number of decompositions are possible. The one presented here, which seems to be the most logical, is based upon the fact that there is only one instruction stream, which is executed by all processors in parallel. The decomposition is: block-structure level, fixed-machine-size level, high-level-language level, and finally machine-code level.

At the fixed-machine-size level, each instruction step where the number of processors in

use is changed is the start of a module. Thus, a module at this level is defined as a sequence of one or more instructions that use a fixed number of processors. As some algorithms execute, the number of active processors is reduced until, at termination, only one processor is active. It seems appropriate to place the fixed-machine-size level below the block-structure level because the number of processors in use is normally changed during the execution of a block.

At the program-execution level, and at the task level, the execution of an algorithm represents one possible path through the object. In some studies a path is divided into two modules: one is the module that solves the problem, and the other is the module that manages parallelism. In practice the execution of these modules will be interleaved, requiring a mechanism for detecting pausing and resumption of modules, if measurements are to be made.

At the machine-code level, a situation similar to that at the micro-code level in a monoprocessor exists. In a single processor machine a single microcode instruction (object) is decoded into a number of parallel operations (parallel modules?); in a SIMD multiprocessor machine a single multiprocessor instruction (object) can be decomposed into a number of parallel processor instructions (parallel modules?). In both cases, if we go below the level of a single machine instruction we have to introduce the idea of parallel objects.

At all levels in the hierarchy, an essential piece of stimulus information is the number of currently active processors. Thus, one additional measure is the trace of changes in the number of active processors: a sequence of triples (number of processors, module identifier, time since start of measurement period).

Two other possible decompositions have been used in reported measurement studies. The first involves decomposing the task level into processes running on separate processors, then decomposing the process level as for a monoprocessor. Decomposing objects in this way is needed when studying MIMD (multiple instruction path - multiple data path) processors, and is discussed in section 9.2.1.

The second possible decomposition reflects the concepts of data flow models of program

execution, and the data intensity of the problems to which parallel computing is being applied. The processing of one data point is a module (i.e. one level below a task), assuming that at least one instruction is needed to process a data point. Thus, a path is a sequence of operations on data. Again, the number of active processors is an important piece of stimulus information. One way of handling this within the proposed hierarchy, is to equate the processing of one data point to the execution of a block at the block-structure level.

The measures, described in chapter 2 for objects on a monoprocessor system, all apply to SIMD machines if the number of active processors is included in some equations. The equations where this needs to be done are identified in the following discussion of the measures proposed by Siegal et al. Those measures not mentioned should apply as they are.

9.1.2. Proposed SIMD Measures

The measures proposed by Siegal et al (1982) are:

- **Execution time** ($T_N(M)$) is a measure of the time involved in performing an algorithm of size M using N processing element. Execution time is the sum of the time spent executing the problem and the time spent managing parallelism. The two main management operations (overhead) are the enabling of processing elements, and the transfer of data over the interconnection network. The definition and measurement of throughput is the same as for single processor machines, however having to consider the sharing of the load across several processors can cause confusion.

- **Speed** ($V_N(M)$) is defined as the number of data points processed per unit time. If the algorithm uses all processors to process one data point then speed is the same as module throughput (N_t) for a monoprocessor at the block-structure level. However, if each processor processes a separate data point then N data points are processed together. The throughput equation (equation 2.30) remains the same, but the calculation of the number of module executions must take into account the number of processors. Thus, the number of module executions is the number of module executions per processor multiplied by the

number of active processors.

- The **Speed Up** ($S_N(M)$) of an N processor algorithm over a one processor algorithm is the ratio of the execution time of the one processor algorithm $T_1(M)$ to the execution time of the N processor algorithm. This is a performance comparison, not a performance measurement, which requires the measurement of the execution times and the number of processors in use. The measurement of the number of processors in use is a natural extension of the formulation to cover parallelism, although, whether the number of processors in use is considered to be event or stimulus information depends on the method of object decomposition, and the level in the object hierarchy. Parkinson and Lidell (1983) question the usability of this measure in a distributed array processor (DAP) environment. The processing elements in the ICL 4096 DAP operate at the bit level, words are handled by paralleling processing elements, under software control. Thus, a single processing element algorithm is a nonentity. Speed up calculations can only be done by comparison to algorithms executing on conventional mainframes, introducing errors into the measure due to the dissimilarity of the processing elements.

- The **efficiency** of an N processor algorithm is the ratio of speed up to the number of processors. Efficiency is a performance comparison, based upon previous measures, which gives a feel for how the achieved speed up compares to the ideal speed up (N). Again, Parkinson and Lidell (1983) question the validity of this measure in a DAP environment and they propose another way of calculating efficiency: the ratio of the number of processors usefully active to the total number of processors. This measure can give a feeling for the quality of an algorithm, but it must be used carefully, because in some associative algorithms (the DAP can be configured as an associative processor as well as an array processor), each step eliminates some of the processors used in the previous step.

- **Overhead ratio** is the ratio of the overhead time to the total execution-time. It is the module-to-path execution-time ratio of the overhead module.

- **Utilization** ($U_N(M)$) is the fraction of time during which the processors are busy executing computations of the algorithm. Assume that for a problem of size M, there are x modules in the N processor computation. Each module(x) uses p_x processors and takes t_x time units to execute.

$$U_N(M) = \frac{\sum_{x=1}^{x} t_x \, p_x}{(N * T_N(M))} \qquad 9.1$$

The information to do this calculation can be obtained directly from the event trace and the stimulus information. This equation is an extension to the formulation of performance measurement required by the decomposition. If the alternate decomposition, that of decomposing to separate processors, was used (as it is in the MIMD case) then the Utilization becomes the summation of the set of module utilizations (U_{mn} *equation* 2.43), if the measurement period is the product of the execution time and the number of processors.

- **Redundancy** is the actual computation time (summed over all processors) divided by the execution time of a single-processor algorithm to solve the same problem. Redundancy is a performance comparison which gives a measure of redundant computations that are performed.

- **Cost Effectiveness** is the ratio of the speed to the cost of the system.

- **Price of the Computation** is the cost of implementing and using a parallel processor to perform a desired computation. These last two measures are performance evaluation calculations based upon measured values.

Parkinson and Lidell (1983) look at two other measures:

- Estimates of **MIPS** (millions of instructions per second), **LOPS** (logical operations per second), and **FLOPS** (floating point operations per second). Commercial estimates of these are usually based upon the reciprocal of the floating-point-multiplication time, and hence, are misleading, because they assume a totally calculation-bound program and they do not take into account word size and number of memory accesses. As with conventional machines, these measurements produce numbers which are valid only for the program being studied. In terms of the formulation of performance measurement, they are measures of module throughput at the machine-code level, where the modules are instructions, logical operations, or floating point operations.

- **Benchmarks** of running programs, which are the same as measuring the execution time.

9.2. Performance Measurement of MIMD Machines

Siegel et al (1982) claim that the measures for SIMD machines will also apply to MIMD machines, however they leave the proof of this assertion to future research. In MIMD machines, each processor has its own instruction stream as well as its own data stream. As mentioned in section 9.1.2 the utilization equation (9.1) is correct for the MIMD object hierarchy. Instrumentation of multiprocessors can be easier than the instrumentation of monoprocessors, because one of the processors can often be used to run the monitoring process. However, the volume of data to be analysed is much larger.

9.2.1. MIMD Object Hierarchy

The hierarchical decomposition proposed for a SIMD machine is not possible on a MIMD machine, because of the multiple instruction-streams. The hierarchical decomposition to be applied to a MIMD object depends upon the way in which the machine is configured. Two configurations will be discussed here, parallel processors and distributed processors, as these represent the ends of the spectrum of possible configurations. The terms parallel processor, multiprocessor, network-of-processors, and distributed processors are often used to mean the same thing by different authors. To reduce confusion, the following definitions are used in this

chapter. In a distributed processor, the multiple instruction streams are separate tasks; in a parallel processor, the multiple instruction streams are all part of the one task. Parallel processors will be discussed in this section and distributed processors in section 9.4.

The conceptual model of an object during execution, that of a sequence of modules, is still the same. The highest level, the system level, decomposes into tasks as in a monoprocessor. However, a parallel task is spread over several interacting processors. Hence, a task is decomposed into a set of programs running in parallel on a set of processors. Thus, a new level is introduced into the hierarchy: the processor level. The hierarchy for a parallel machine is: system, task, processor. Below the processor level, the hierarchy is the same as for a monoprocessor, and thus the measures are the same.

In a parallel processor, process interaction is of major importance. Below the task level, we are measuring tasks on individual machines, and the interaction with other machines can be described in terms of external events. This is analogous to the interaction between processor and peripherals on a monoprocessor. These external events will cause changes in either the execution path or the stimulus information. Thus, they can be detected in the standard measures. The number of active processors is an important piece of stimulus information, at the task level, particularly as we will be recording an event stream for each processor. Parallel event streams, and their interactions, can be displayed on an event-trace graph (figure 2.3).

An alternative way of looking at a SIMD machine is to see it as a MIMD machine where the parallel instruction streams are identical. Thus, the SIMD could be considered as a special case of the MIMD.

In the next section measurements reported on MIMD machines are discussed within the context of the extended measurement formulation.

9.2.2. Measurement of Parallel Processors

The **Cm* machine** (Gehringer et al 1982) is a network of clusters of processing elements which can be configured as a parallel processor (called a multiprocessor by Gehringer et al), or as a

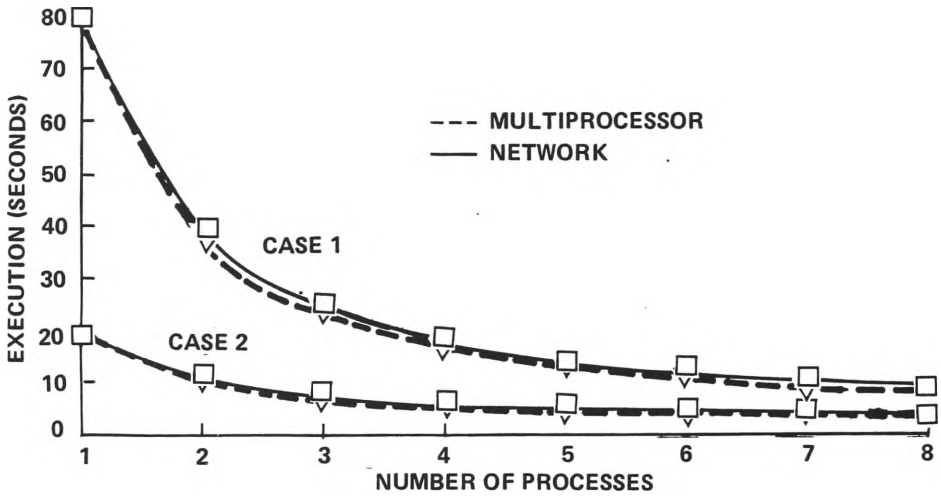


Figure 9.1 Comparison of integer programming on network and multiprocessor configurations of Cm*, with two data sets (Gehringer et al 1982)

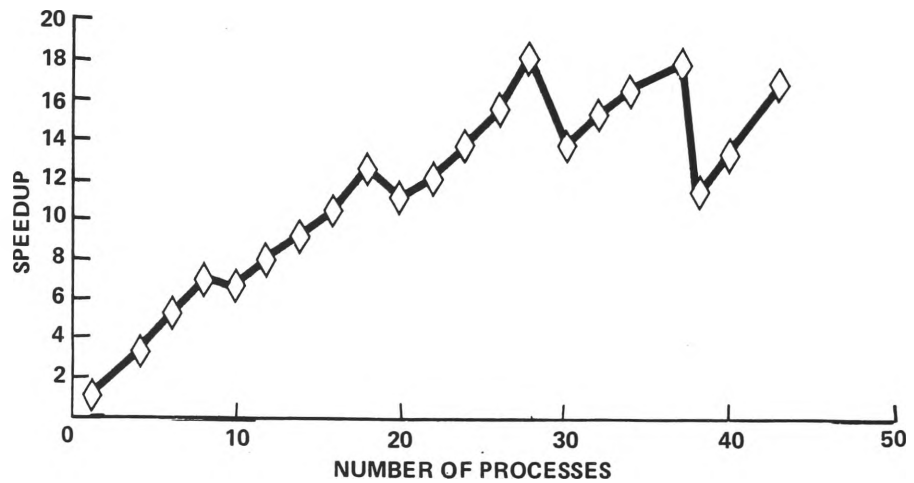


Figure 9.2 Speedup for the quicksort algorithm on Cm* (Gehringer et al 1982)

distributed processor (called a network of processors by Gehringer et al), where each processing element can only access its own portion of memory. Some experiments have been performed to compare the execution of algorithms in both configurations. The comparison (figure 9.1) compared the execution times of the algorithms on both processor configuration with a variable number of processors and different data sets. Again these measures are in accordance with the formulation of performance measurement, with an extension to allow for the number of processors.

In order to study whether parallel processors can perform a substantial amount of computing more efficiently than alternative architectures, speed up, as defined for SIMD machines, was measured during a variety of experiments, and plotted against the number of active processors (figure 9.2). Experiments were set up to evaluate: optimum processor configurations for solving different problems, the overhead penalty, the cost of synchronisation, and the trade-off between locality of reference and memory contention. One algorithm, a railway-network simulation which could only run on a parallel processor configuration larger than three processors, was studied using execution time, because the speed up calculation could not be performed due to the lack of a uniprocessor solution.

The **Erlangen general purpose array processor** (EGPA) is a three dimensional processor array which has the structure of a pyramid (Fromm et al 1983). The elementary cell consists of a processor and a memory; the elementary structure is a pyramid having 1 top and 4 bottom cells. The machine can be configured so that either a processor can execute a separate job, or a large job can be distributed over several processors. The latter mode leads to a control flow with dynamic processor-processor interaction that is not easy to understand.

The EGPA processor includes both hardware and software performance monitoring tools. These tools were designed on the basis of the following principles of measurement, which are consistent with the formulation of performance measurement. All processes are assigned different process numbers, which can be used as priority numbers. When a process is active, its process number is stored in a hardware priority register. A hardware measurement recording

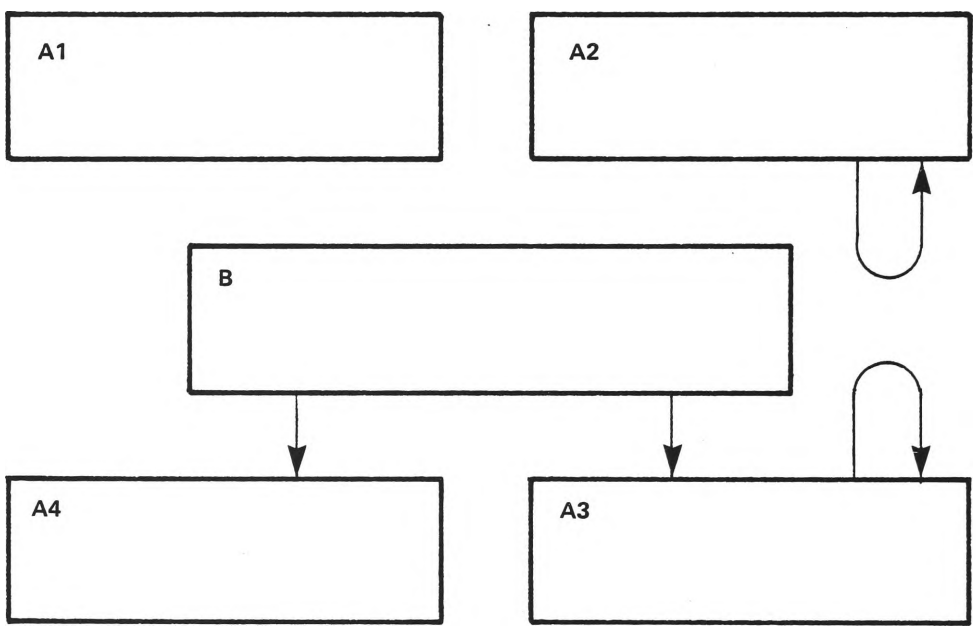


Figure 9.3 Graphical display of simultaneous events on five EGPA processors (Fromm 1983). Each box represents a processor and internal events. Arrows returning to a box represent I/O operations. Events concerning two processors are indicated by arrows between boxs

the flow of process numbers in the priority register results in an exhaustive, and precise, description of all activities at the process level. An ordered set S of sequence steps s_i.

$$s_i = (p_i, t_{pi}) \qquad 9.2$$

where i = 1 .. n

is the result of the measurement, where p_i is the number of the process which has been active in the ith sequence step, and t_{pi} is the time the process spends in the ith sequential step. Thus, the resulting trace is an event stream which contains both the sequence in which objects are executed and their execution times. When studying the execution of a job distributed over several processors, it is important to be able to determine the concurrency of the processes in different processors. This is done by correlating the arrival times of the sequence steps in the independent, one for each processor, event streams, i.e. each sequence step is effectively time stamped relative to a master clock.

The software tool produces a similar event trace based upon the execution of event identification marks (monitor subroutines) placed in the programs. Both types of event traces can be walked through on a graphical display (figure 9.3). Interaction, and correlation, between parallel streams is also shown. An interactive program plots the distribution of the duration of interesting events (an execution-time profiler).

Franta et al (1982) extend their model of a sequential program (described in section 3.6) to a model of a concurrent program. An execution history is defined for each processor as if it were a single processing unit. These histories are then composed into a set of histories (a hierarchical composition) of the multiprocessor. At this point the model does not cover interactions. Individual program-histories are interleaved to make a single event-history, on the basis of the ordering of data transfers, via four communication primitives, between processes. This results in a single history of the changes to the data in the system. The model entertains no notions of a global clock, but one has to be introduced to make instrumentation possible, almost as an afterthought. This model is usable if you are only interested in what happens to the data, and the ordering of data interactions between processes, but it has no concept of program flow. Thus,

it may be suitable for describing a data-flow machine, but it is incapable of providing the measures discussed for a SIMD machine.

9.3. Instrumentation of Parallel Processors

In a number of research machines, performance-measurement instrumentation has been included in the initial design. The trend is to integrated instrumentation-environments (Segall et al 1983), also called distributed-system test-beds (Franta et al 1982), which support: measurement of performance; execution and control of experiments; specification of synthetic work loads; control of the interprocessor communications network; and, in some cases, control of the master system clock; to the point where an experiment can be stopped in mid flight for examination of machine state.

Control of integrated instrumentation is usually centralised in an experiment managing program, which runs on a separate processor. The experiment management processor can be one of the parallel machine's processors not being used in the experiment, or a processor independent of, but connected to, the parallel processor. Hardware and software tools added to the processing elements, detect events, record the state of the processing element (and its processes) at the time of the event, and transfer the state information to the experiment manager.

Information recorded by hardware tools is often transferred over a separate instrumentation bus to the experiment manager. The interface between the instrumentation bus and the processor may include event filtering, and sequencing, circuitry. Event-filtering circuits can be similar to the plug-board front-ends of traditional hardware-tools, but are normally closer in concept to the front-end of a logic-state analyser. Event sequencing circuitry ensures that the correct time sequence of the events occurring in the concurrent processes is maintained when the data is read from the parallel event streams and recorded.

Information recorded by software tools can either be written to the instrumentation bus via hardware probes (hence forming a hybrid tool) or transferred to the experiment manager

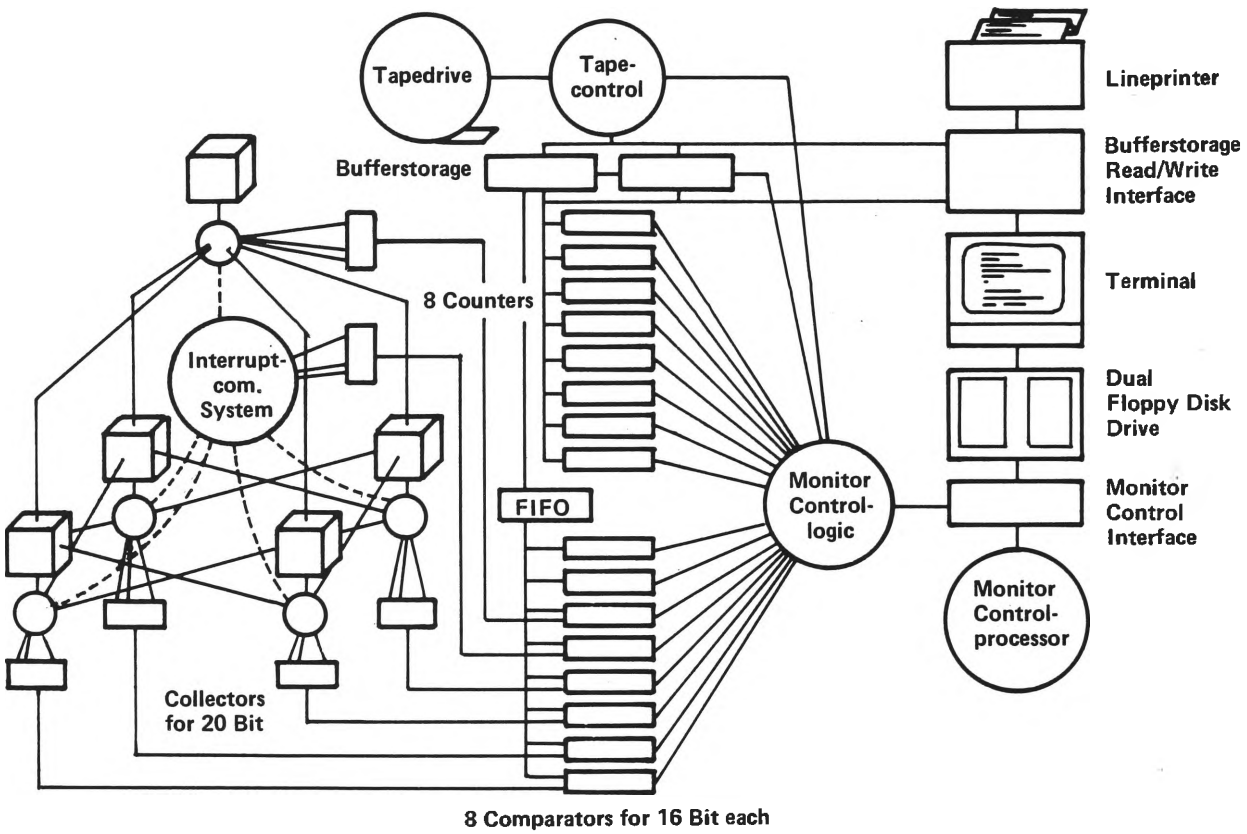


Figure 9.4 Block Diagram of the hardware monitor of the EGPA pyramid (Fromm 1983)

over the parallel processor's interconnection network. As with conventional monoprocessors, software tools cause some interference to the system, and hence some degradation in accuracy. These techniques are illustrated in the description of the instrumentation of two significant research processors, that follows.

9.3.1. Instrumentation of the EGPA Pyramid

The EGPA pyramid is instrumented by two fully integrated measurement tools (Fromm 1983):

- a hardware monitor, which is connected by electronic probes to every processor (figure 9.4), and
- a software monitor which is an integral part of the EGPA environment.

These tools demonstrate the application of a subset of the formulation of performance measurement to the measurement of parallel processors - my comment not Fromm et al's.

To combine the event traces from asynchronous processors into a single correctly ordered trace of events two techniques are used; one for software measurements and one for hardware measurements. Software measurements are based on periodically enforced clock synchronisation in all processors. Thus, time stamps can be used by the evaluation program to determine the ordering of originally independent results. Hardware measurement are fed to a set of simultaneously operating comparators whose results are joined by a hardware FIFO unit.

Fromm et al were interested in gaining insight into the dynamic flow of activities in the pyramid, and into what was going on in the different processes at the same time. They decomposed the task being executed (object) into processes (modules) and recorded the stream of processes (module trace). Normally, only software measurements can recognise which process is active at a given time, but the operating system assigns unique process numbers to all processes, which are stored in a hardware register by the scheduler. From the register they are recorded by the hardware probe, every time the register is updated. Thus, the overhead costs of a special software probe are saved. This highlights the gains which can be made by integrating measure-

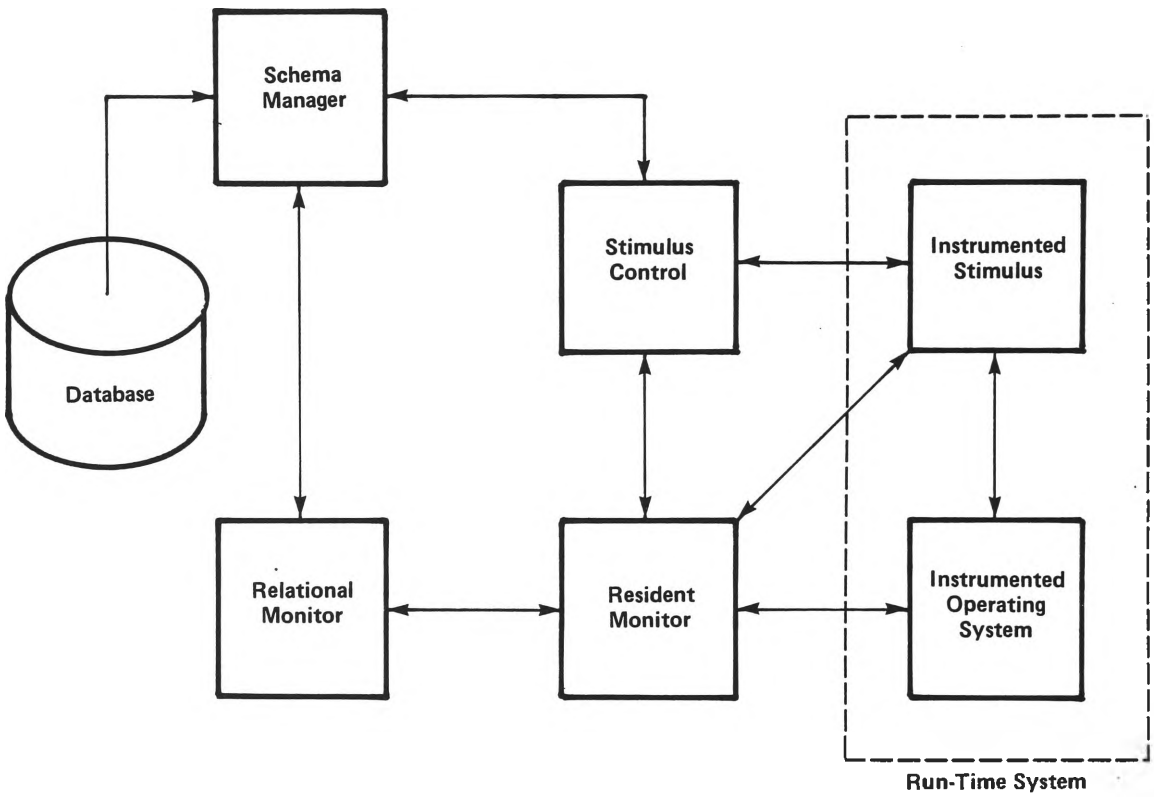


Figure 9.5 Cm* Integrated Instrumentation Environment (Segall et al 1983)

ment tools into the system at design time.

The software instrumentation records user oriented event traces; in contrast to the system oriented event traces recorded by the hardware instrumentation. Software probes (marks) store event-descriptors (markers); consisting of identification of the place in the program (mark) from which the probe was called, real time, and process CPU-time (both in multiples of 50μsec); in a marker buffer. System marks, usually pairs of entrance and exit markers which are permanently inserted at strategic points of the programming environment, inform the applications programmer of essential events in a symbolic form. User marks, placed in the applications program by the programmer, can be selectively activated prior to program execution.

Parallel event streams are converted to sequence pairs (event, duration) and combined in real-time order into a single stream by software. Since events from different processors overlap, they are displayed graphically (figure 9.3). Measurements made with these tools have been used to validate a queueing network model, to study memory contention problems when several processors access one processor's memory, to study the synchronisation of communicating processes, to investigate why the synchronisation time was long, and to test synchronisation time reduction hypotheses.

9.3.2. Instrumentation of Cm*

The Cm* integrated instrumentation environment (Segall et al 1983) contains a set of tools (figure 9.5) which enable the generation and measurement of experiments, where an experiment is the execution of an instrumented program in a controlled environment. The environment contains several components: a schema manager, a run-time environment, an instrumented stimulus, an instrumented operating system, a data base, and a monitor. A schema is a complete experiment description consisting of an application program, or synthetic work load (stimulus or task force), to be measured, monitoring directives, system configuration information, and experiment directives. The results of an execution of a schema is captured in a schema instance; containing measurements, values of schema parameters, and environmental

information.

Software sensors, which reside in the operating system and resident monitor, and may reside in the stimulus, generate event records. An event record contains an indication of the operation being monitored, the name of the component performing the operation, and the name of the object the operation is being performed on. The event record may also contain a time stamp and other information about the event. Some sensors are automatically built in, for example at the start and end of sub-tasks, to record timing information. When a sensor detects an event, an event record is placed into a data structure, one for each instrumented object, called a receptacle. Event filtering is done by resetting an event enable flag in the receptacle. If the flag is set, the resident monitor extracts the event record from the receptacle and sends it on to the relational monitor.

One of the tools developed to aid the rapid development of task forces (experiment directives), a work load generator written in a high-level behaviour-description language, is used to specify the behaviour of the parallel program and the placement of sensors. When the work-load generator is translated, sensor descriptions and data structures, which allow the stimulus controller to exercise external control over the experiment, are generated. Another program takes the sensor description and generates optimized code for each software implemented sensor.

The relational monitor collects the event records passed to it by the resident monitor, and builds a schema instance, which is stored in a relational data base. Information is recorded in tables of tuples, called relations. In the **running relation** each tuple records the occurrence of a particular event. A **period relation** records a relationship that exists for a period of time. Periods are delimited by events; each tuple (period) in the running relation is associated with two event tuples: one in the **start relation** and one in the **stop relation.**

Primitive relations, such as those above, contain information that is a direct translation of a set of recorded events. New relations called **derived relations**, can be defined as a result of opera-

tions performed on existing relations. Derived relations are specified using a query language, and can be calculated by the relational monitor if requested in the schema. After the experiments, the user can perform analyses across schema instances using standard enquiries on the data base.

This represents a very ambitious attempt at instrumenting a system. Segall et al (1983) indicate that at the time of writing it had only been partly implemented. The type of data recorded in the event records by the sensors appears to correlate with the measures specified by the performance measurement formulation.

9.4. Performance Measurement of Distributed Processors

In a distributed processor, each processor works on a different task. A task is not spread over several processors, but is contained within one processor. Tasks may communicate with one another over a network, which interconnects the processors. Often, scheduling of tasks is done by one supervisory processor. Thus, at system level, a distributed processor is considered to be one machine.

Our conceptual model of an object is still a sequence of modules. In the distributed processor case, the system decomposes into processors. So the hierarchy for a distributed processor is: system, processor, task. Below the task level, the hierarchy is the same as for monoprocessors. The amount of coupling between tasks depends upon the job being executed by the system, and the scheduling algorithm. Tasks may be completely separate; different user, different job class, different data; or they may be loosely coupled. To perform a user's request, a task running on one processor may be pipelined with a task on another processor, e.g. compile on one, execute on the other.

Below the processor level we again have parallel event streams, but their interaction, if any, may be of no interest to the analyst. The number of active processors is an important piece of stimulus information at the system level. The instrumentation of PRIME and C.mmp are discussed in the following section.

9.4.1. Instrumentation of PRIME

PRIME (Ferrari 1973), a computer system developed at the University of California, Berkeley, was designed to provide interactive service to a number of terminals with very high degrees of availability and privacy. PRIME consists of five processors which are linked together, and to peripherals, by an interconnection network. Each processor can access roughly eighty per-cent of the available memory, with each 8K memory module being accessible by three processors. A high level of interactive performance requires that resources are dynamically allocated to the processes competing for them, but the high-level of privacy requires that different processes are not allowed to share the same memory module.

The hardware is partitioned into five physical subsystems, one of which carries out centralised operating system functions, and the other four handle user processes. No resource sharing is allowed between the four user subsystems, and they communicate with each other over the interconnection network. Message buffering is done by the central system.

The design of instrumentation for PRIME was impacted by the distributed architecture and the privacy requirement. Some variables, for example traffic rates on the interconnection network, could not be measured in the central system. Thus, instrumentation had to be distributed. Collected data could not be sent over the network to a central point, because it would degrade the performance of the system if the data rate was high and the network bandwidth low. Thus, each processor has to be probed separately, and a separate data collection network added to carry the data to a hardware monitor.

The privacy requirement means that one subsystem cannot be used to measure another, unless the measured process collects the data. In order not to violate the privacy requirement, measurements can only be made of the virtual subsystem created for the user. The system manager could measure system activities, but not user activities.

Both general-purpose and special-purpose tools were included in PRIME (Ferrari 1973). Three general-purpose tools were developed: a very limited, inexpensive hardware tool; a

powerful set of firmware tools; and an extensive facility for general-purpose software measurement. Users are allowed to use the software tool within the privacy of their own virtual subsystem, but only the system manager is allowed access to the hardware and firmware tools.

The firmware tool consists of two probes: one periodically samples the contents the microcode program counter; the other detects the execution of conditional-branch instructions and generates a flag indicating whether the branch condition was satisfied or not. The information collected by the first probe is used to derive utilization factors for microcoded operating system modules. The information collected by the second probe is used to reconstruct the microprogram's flow of control. The data produced by these probes is collected by a hardware tool, which has access to a number of registers in the computer (one on each processor). The sampled addresses and the branch trace are recorded in a memory module. Data from all processors is multiplexed into the one memory module.

The software tool (Ferrari and Liu 1975) supports three measurement techniques: checkpoints, sampling, and tracing. User written measurement routines, which have been checked against correctness criteria, can be called by checkpoints inserted into the user's program. Event counting checkpoints, event tracing checkpoints, and measurement routines are provided. Samples are taken by a software tool, which interrupts the user process at regular intervals. Tracing is based upon event detection checkpoints. All of the tools can be added, removed, enabled, and disabled by simple interactive commands prior to program execution. During execution of the program, program instrumentation cannot be changed. Data collected by the measurement routines is saved on a file for later analysis.

Insertion of checkpoints into a program involves replacing a program instruction with a checkpoint call instruction - a patching technique similar to breakpoint insertion techniques of assembler debuggers. After execution of the measurement routine, the replaced instruction must be executed before flow of control is returned to the program. Two ways of connecting checkpoints to measurement routines were available. In the first, a software switch, the checkpoint calls a routine to save the machine state and then calls the measurement routine. In the second,

a user-defined operation-code calls a measurement routine when ever it is executed. When a user-programmed operation-code is executed, the program counter is saved and flow of control branches to a pre-defined location. The operand field of the instruction can be encoded to specify which measurement routine is required. Switches cause less time interference and user-programmed operation codes cause less space interference.

Special-purpose tools consisted of hardware, firmware, and software meters and logs. These are ad-hoc tools, i.e. tools designed to detect a pre-defined event or a small class of events. Ad-hoc tools can only be included if measurements can be specified accurately at design time.

9.4.2. Instrumentation of C.mmp

A hardware monitor (Fuller et al 1973) was used to measure the C.mmp multi-miniprocessor, developed at Carnegie-Mellon University for research into multiprocessors (Wulf and Bell 1972). Each PDP-11 processor in C.mmp is a complete computer with its own primary memory, controllers, and peripherals. A cross-bar switch allows connection between the processors and shared memory on a per reference basis, and resolves conflicts between such requests.

Questions of particular interest to the researchers who built C.mmp related to the:

- interference in the switch connecting processors to memory,
- configuration of compilations on the multiprocessor - which configuration (parallel, pipelined, network, distributed) suits which class of problems, and
- efficiency of the decomposition of an algorithm into separate processes.

The hardware monitor is a peripheral connected between two Unibuses, one bus in the monitoring processor and the other bus in the target processor. As all communications between processor, memory and peripherals occurs over the Unibus, it conveniently concentrates most of the signals of interest. A few signals internal to the processor are also connected to the hardware monitor. A primitive event detector senses events at the Unibus cycle level. An event

accumulator counts the occurrence of primitive events, and saves the bus information when a predetermined count is reached. The event record includes the Unibus signals, other signals connected to the monitor, and a time stamp. Four primitive event detectors can run simultaneously.

The event detectors can be used to detect entry to and exit from a routine of interest, for example a communication routine, and count the execution time of the routine, or the number of instructions executed. The monitor can also measure the contention for memory by several processors by measuring the increase in memory access time.

10. Other Measurement Applications

In this chapter, a number of applications not discussed in previous chapters are examined in the context of the formulation of measurement. Each application requires the design of measurement tools, and experiments, which are tailored to obtain information peculiar to that application. The concepts of objects, object hierarchy, sequence of modules, event trace, and stimulus information are used in each case.

The first step in designing a tool for a specific application is to understand the application. Then, the purpose of measurement in the context of that application can be clarified. Once the object to be measured has been defined, it is decomposed into a sequence of modules, it is instrumented to produce the desired event-trace, and the exact set of stimulus information to be recorded is determined. Selecting the set of stimulus information is often the most important, and most difficult, part of tool design, as it requires a thorough understanding of the application.

Tailoring the general measurement tool to a specific application may also involve modifying the object data-structure, and hence the data reduction and analysis algorithms. For example, we may desire to record several distinct sets of stimulus information, or to count the number of occurrences of a specific stimulus variable.

10.1. Performance Models

Much scientific research consists of developing models of the system we want to study, and carrying out experiments upon those models. Models are simplified abstractions of the systems we want to study. Unnecessary details can be removed, and only important parameters analysed. Models can be used to reduce a complex problem into a simpler problem, which is more easily understood and is mathematically tractable. Thus, modeling is a way of simplifying a system, and at the same time highlighting the important elements of that system. Detailed models give more accurate results than simple models, but can be more difficult to understand and impossible to describe mathematically.

Models have some advantages relative to direct measurement: parameters and inputs can be changed at will to study system behaviour without the problems this would cause on a real system, and models can be used to predict the impact of system changes on performance without having to make the changes. However, models are not as accurate as measurement (as they are an abstraction and simplification of the system), and measurements must be made on an actual system to calibrate the model and to validate the model's results (Spirn 1977).

A number of different models are in common use in performance evaluation, some highly detailed and some more abstract. Models of objects tend to become more abstract as you move up the object hierarchy. At low levels in the hierarchy, detailed models are intellectually manageable because of the limited number of details. At higher levels in the object hierarchy, the level of abstraction is increased in order to keep the model intellectually manageable.

The current state of performance modeling is that we have a collection of modeling techniques, each suitable for modelling at a different level (or range of levels) within the object hierarchy. The choice of the model to be used, at a particular level in the object hierarchy, depends upon the accuracy required and the cost you are willing to pay. At higher levels in the hierarchy the cost of accuracy increases. Kumar and Davidson (1980) suggest that ... *judicious use of a hierarchy [of models] can simultaneously satisfy the conflicting needs of low complexity and high accuracy ... By using a performance model hierarchy, the system analyst enjoys the tractability of analytical studies while achieving the accuracy of empirical studies.*

Current modeling techniques are clearly defined, often mathematically, and well understood, but if they are to co-exist in a hierarchy of models further work needs to be done on the interrelations between these models. What is needed is an underlying formulation of performance modeling which can be used to codify the models into a unified body of knowledge, in a similar manner to which performance measurement has been codified in this volume. Much work has already been done in this area (Courtois 1977), and as we shall see, the conceptual model upon which the formulation of performance measurement is based fits into such a model hierarchy.

An interesting area of research is the study of whether one type of model can be used at all levels in the computer system. As we have seen, the conceptual model of an executing object can be applied at all levels of the hierarchy, and hence the one set of measures applies at all levels. Is this also true of other models? The concept of near-complete decomposability provides a theoretical basis for a hierarchical modelling approach, and has been applied to queueing-network models with encouraging results (Courtois 1977). For a hierarchical decomposition to work well, the degree of interaction among components within a module should be high with respect to the degree of interaction between the module and other modules in the object. Also, can our models be transported from one system to another dissimilar system, or are more theoretical advances required before this can happen?

10.1.1. Types of Models

Svobodova (1976a) divides system models into three general classes. A **structural model** describes the characteristics of individual system components and their interactions. A **functional model** describes how the system operates such that the model can be analyzed mathematically or studied empirically. A **performance model** is derived by the analysis of a functional model, or a structural model, for a specific model of system workload.

Examples of structural models include block diagrams, some modeling-language descriptions, and some detailed simulation models. These models show the paths of data flow, and control flow, but do not specify why particular paths are followed. Thus, they are more useful for understanding how a system fits together than for understanding how it works.

A number of functional models are in use. They reflect different conceptual models of object execution, and are used to study different system characteristics. A **flow-chart model** is a directed graph where the nodes represent computational tasks and the arcs show the possible flow of control between tasks. Flow-chart models are used for studying program efficiency, and program execution-time. The object flow-graph, defined in chapter 2, is a flow-chart model based upon actual measurements. Thus, the formulation of performance measurement takes a flow-chart model and applies it to all levels of the object hierarchy. Having obtained

the desired model by measurement, performance can be evaluated by using the model to predict the extent of the object for synthetic workloads.

A **finite-state** model is also a directed graph, but the nodes represent the state of the system and the arcs represent transitions between those states. These models can be used to represent concurrent operation, and to analyse the utilization of resources. A model of this type can be developed from a system object-record if appropriate stimulus information is recorded. Stimulus information is used to calculate the probability of a certain state transition given a certain workload.

In a **queueing model**, a computer system is a set of resources and queues for those resources. These models emphasise the flow of jobs through a system in terms of the time taken by the resource to service the job, and the time the job waits in the queue for the resource. As queuing models are very widely used, measurement for queuing models will be looked at in a later section.

Two classes of **performance models** are in common use: **analytical models** and **empirical models**. An **analytical model** is a set of mathematical equations which express the relations which exist between the basic system variables and performance parameters. Analytical models are limited to simple system models, for example queueing models, or detailed models of one resource, by the requirement of mathematical tractability. Thus, the underlying functional model must meet the conflicting goals of capturing the basic structure of the system, having the characteristics of a real work-load, and be simple enough to be mathematically tractable.

An **empirical model** is developed by analysis of empirical data, often by statistical methods, for example a regression model. A regression model is calibrated to match one set of observations, and thus may only be valid for a limited work-load-range on one system. Unless the model captures the basic elements of the system, possibly due to having an analytical basis, it is not transportable. A second type of empirical model is the system, or resource, or object, profile. A number of these are shown in chapter 2 as graphical methods of data analysis.

10.1.2. Measurement for Models

Measurement plays an important part in modeling system performance. To comprehend the applications of measurement in modeling, we will now look at a typical modeling methodology.

A performance analyst first defines the goals of the modeling study: what problem is to be studied, is there a suspected performance bug, is there a need to upgrade the system to handle a larger work load, etc. From these goals, the analyst can decide at which level in the object hierarchy the system is to be modeled, and select an appropriate modelling technique. The assumptions upon which that modeling technique is based should also be delineated, for example certain time distributions are assumed in queueing models.

As with measurement, understanding the system to be modeled is important. The analyst develops a model of the system within the constraints of accuracy and cost. Having developed the model as a set of equations (or graphs), the model must be calibrated and verified. The analyst has to define: what to measure, and how to measure it. **Calibration** of the model involves using measured data to calculate model constants. Thus, data must be collected from the system when it is executing under conditions similar to those being modeled.

Once the model is calibrated, it must also be verified. **Verification** involves stimulating both the model and the system being modeled with the same inputs, and then comparing the outputs to see if they agree. If they differ significantly, then the model has to be updated. The processes of model calibration, model verification, system understanding, and system debugging often interleave and interact. A disagreement between model output and system output may well be due to a system bug. In this situation, studying the system to update the model will lead to understanding what is wrong with the system.

Once the model has been calibrated, and verified, it can be used to study the impact of varying stimuli on system performance. Also, it can be used to predict the impact of an upgrade on system performance. Once the upgrade is done, the system is again measured to check the validity of the predictions. Model predictions will be inaccurate if a significant parameter has been left out. This parameter may not have been significant in the initial system

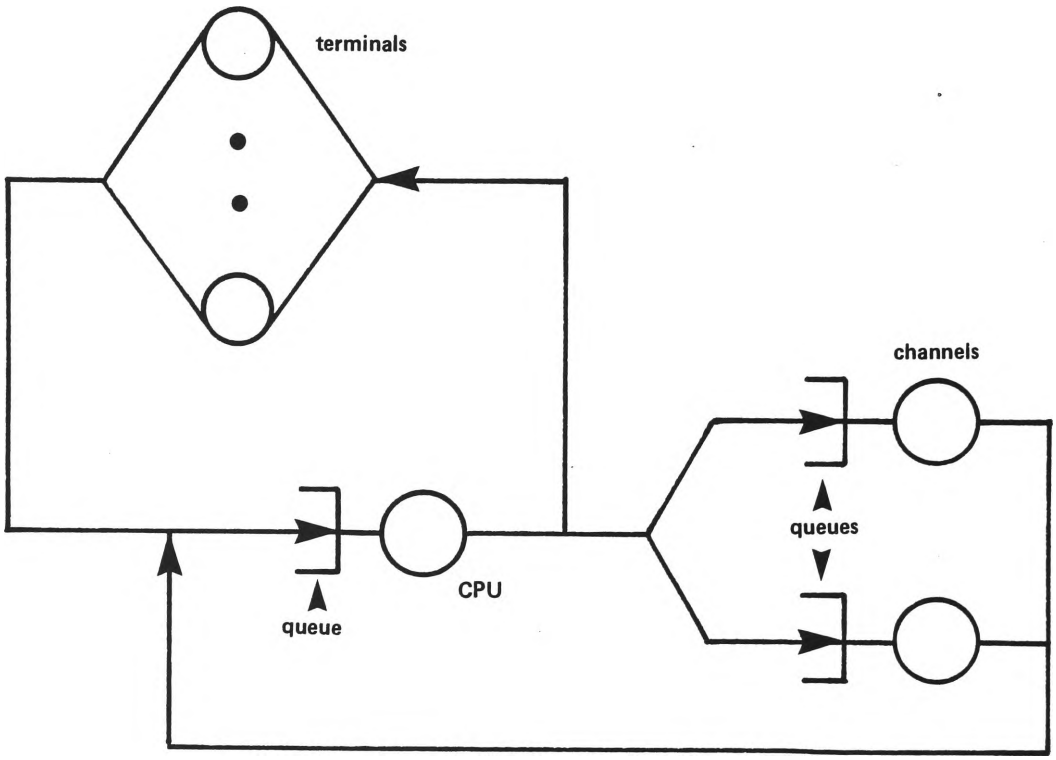


Figure 10.1 Typical Queueing Network Model (parameters given in table 10.1)

model, but new stimuli, or a change in system configuration, may produce a situation where it becomes significant. Such situations are detected by comparison of measured outputs and modeled outputs.

Thus, measurement plays a significant role in performance modeling. In fact, modeling is a very poor science if it is not based upon accurate measurement.

10.1.3. Measurement for Queueing Models

Most books on performance modeling include an overview of measurement (for example Sauer and Chandy 1981) but rarely give a detailed description of how to obtain the data for the models discussed within the text. Many models have been developed for IBM systems, and the data is usually obtained from the files produced by accounting programs, for example the System Management Facility (SMF) and the Resource Measurement Facility (RMF) on the MVS operating system. Unfortunately, these programs do not record all the desired data, and some data has to be estimated from the recorded data (Graham 1981). The most comprehensive study of measurement procedures for queueing network models, including a survey of tools on a variety of systems, is found in Clifford Rose's (1978) paper. Tolopka (1981) discusses an event-trace monitor used on a VAX 11/780 to collect data for use in a queueing model.

The XRAY monitor (Blake 1980) monitors the performance of a network of TANDEM/16 computers to detect bottlenecks: the overuse of hardware components, and the software source of the associated activity. Placement of software meters in the GUARDIAN operating system was guided by a simple model founded on operational analysis (Buzen 1976). The model has the following general form: system throughput, in transactions per second, is given by the ratio of the utilization of a device to the demand for that device per transaction. From analysis of the model Blake concluded that measurement of device utilization and visit rates would provide all the data needed for bottleneck identification. The system was instrumented accordingly, and then the model was used in the analysis of the collected data.

Queueing Network models (figure 10.1) are based upon the following conceptual model of

a computer system: a number of jobs of various work load classes competing for a limited set of resources. The system is decomposed into a set of resources for which jobs must queue if they want service from a resource. Even though this conceptual model differs from that used in the formulation of performance measurement, instrumentation based upon the formulation can be used to measure the model parameters. The data collected by the instrumentation will be more detailed than that normally used in models, in the sense that it is possible to measure the parameters for an individual job. However, measurements for a set of individual jobs can then be averaged to get the data for the model. Alternatively, modeling can be extended from looking at the average job to looking at individual jobs.

A system can be decomposed into a set of tasks (jobs). Each job is decomposed into a set of modules. This set is the union of four subsets:

- modules that add jobs to resource queues,
- modules that wait on resources,
- modules that remove a job from a resource queue so that it can use the resource, and
- modules that use resources.

Thus, the events we wish to detect are:

- the addition of jobs to a resource queue,
- the removal of jobs from a resource queue,
- the acquisition of a resource by a job, and
- the termination of resource use by a job.

These events can be detected by placing probes into the routines which handle the queues, and into the routines which dispatch jobs to a particular resource. These probes generate the event trace. In addition, stimulus probes must accompany all event probes. When a job is initiated by adding it to the scheduler queue; the job number, the job class, and the current length of the queue must be written to stimulus probes. During the lifetime of the job, and at termination, the job number, and the length of the queue, must be output as stimulus information whenever

a. Measured Parameters

A	number of arrivals
B_i	time server i was busy
C	number of completions
C_i	number of completions from device i, i = 0 is the system
K	number of devices ≡ number of users who can receive simultaneous service
M	number of users ≡ number of interactive terminals
N	maximum number of jobs in the system ≡ multiprogramming level
n_{ri}	number of class r jobs at device i
S_i	mean service time at device i
T	length of the observation interval
V_i	number of visits to device i per job
W	accumulated time in the system - job seconds = $\int A - C\ dt$
Z	mean think time

(b) Calculated Parameters

λ	arrival rate = A/T
μ	service rate = $1/S_i = C_i/B_i$
n	mean number in system = W/T = XR
R	mean time in system = W/C
S_i	mean service time = B_i/C_i
U	utilization = B/T = XS
V_i	visit ratio = $\frac{C_i}{C_o}$
X	throughput = C/T

Table 10.1 Parameters of a Typical Queueing Network Model

the job is added to, or removed from, a queue. The job number must be output as stimulus information whenever it acquires or releases a resource.

Additional probes are needed in terminal handlers to measure the number of active terminals and the think time. Think time (figure 1.1) is the time between when the computer completes the execution of a user command and when the user completes the entry of a new command. Normally, measuring think time will require the instrumentation of the particular interactive program because only the program knows when it has completed the command.

From the event trace generated by the above probes, all the information needed to calculate the parameters for a typical model (table 10.1) by job class can be derived. Many of them have direct mappings into values in the object record, for example, cpu service time maps to module execution time, and visit counts to module execution counts. A modified object record, and hence a modified collection algorithm, would make measurement for queueing network models easier. The object record would be modified to store the additional stimulus information. The data collection algorithm would be modified to calculate the actual parameters, and to give the modeler the option of collecting data for only one job class, or even for only one job etc.

10.2. Man-Machine Interaction

As a general rule, measurement of human performance during interaction with computer systems is not viewed as a normal part of the software development process (Dunham and Druesi 1983). Yet, how well the users accept a system ultimately determines its success. With the current trend toward user friendly interfaces, for example, the 'Small talk Environment' as implemented on the 'Apple Macintosh', measuring and modeling human performance during interaction with computer systems must become a significant part of software development.

The enormous variability among people makes comparison of the effectiveness of various software tools, and hardware input/output devices, very difficult. Measurements of the time taken by programmers, with widely varying experience, to find bugs introduced into Fortran

programs show no correlation with experience (Sheppard et al 1979).

Some people find learning new techniques very difficult and it may take a long time for their productivity on a new, user-friendly device to surpass that on an old awkward, but familiar device. Some people think with visual models, others with verbal models, and still others with numerical models. Each will prefer a different interaction style. Our previous experience often flavours our attitude towards a new way of doing something, for example people who have learned to program in a spaghetti-code language find programming in a structured language requires a very different mental approach. Sometimes the new device requires the same steps to be done in a different order making it confusing to use, for example someone who has learned to use an algebraic calculator finds a reverse-polish calculator difficult to use and vice versa.

When attempting to tailor a user interface to a specific task, or to a specific level of user experience, attention must be given to:

- the choice of dialogue style,
- which input/output device suits the application,
- the consistency of command syntax, and
- choosing an interaction sequence that is appropriate to the task, suitable to the user, and predictable.

To do this the designer needs to understand user characteristics. Carey (1982) has grouped the user characteristics that impact upon user acceptance of systems into three categories:

- Differences in the nature of the task to be performed and the associated mental model.
- Differences in the nature and extent of the user's cognitive model of the information system. Some users approach a given system already equipped with a conceptual framework within which a system model quickly develops; others may use a system regularly without developing more than a procedural understanding of it. Also, the user's cognitive style influences the nature of his conceptual model.

- Differences in the nature and extent of the user's exposure to the system.

One method of tackling the problem (Pfaff et al 1982) of designing a standard interface for a variable set of individual users is to define user interaction in terms of a model consisting of a small set of well defined logical input and output devices. All input/output to applications programs is coded in terms of these logical input/output devices. An interface program is used to map these logical input/output devices to different physical devices, different user classes, and different applications. The Graphical Kernal System (GKS) has been designed using this approach.

A complementary approach is to direct all user interaction through a common input/output program, which individual users can tailor, to some extent, to their own desires. This common input/output program presents a consistent user interface at all levels of system interaction (Nievergelt 1982), eliminating the annoyance caused in many systems by one command meaning different things at different places in the system.

When faced with the great variability of users, it is tempting to forget measurement and use 'gut feel', or the subjective comments of a sample of users, as a guide in determining the effectiveness of an interactive dialogue. Whether the method of interaction has improved productivity or not can only be determined by measurement.

Two things that can be measured easily (compared to the difficulty of measuring human characteristics) are system response time (Miller and Thomas 1977) and user response time (English et al 1967). A typical user interaction sequence is shown in figure 1.1. Most reported measurement studies have been in these two areas. The following examples are typical of the work done in these areas.

R.B. Miller (1968) gives possibly the best conceptual analysis of the effect of system response time on user response time, and suggests maximum system response times for a variety of user input situations. Appropriate and timely feedback to the user of meaningful information is one of the keys to successful interactive dialogues. L.A. Miller and J.C. Thomas (1977) give an overview of the behavioural issues involved in interactive systems, and how these can

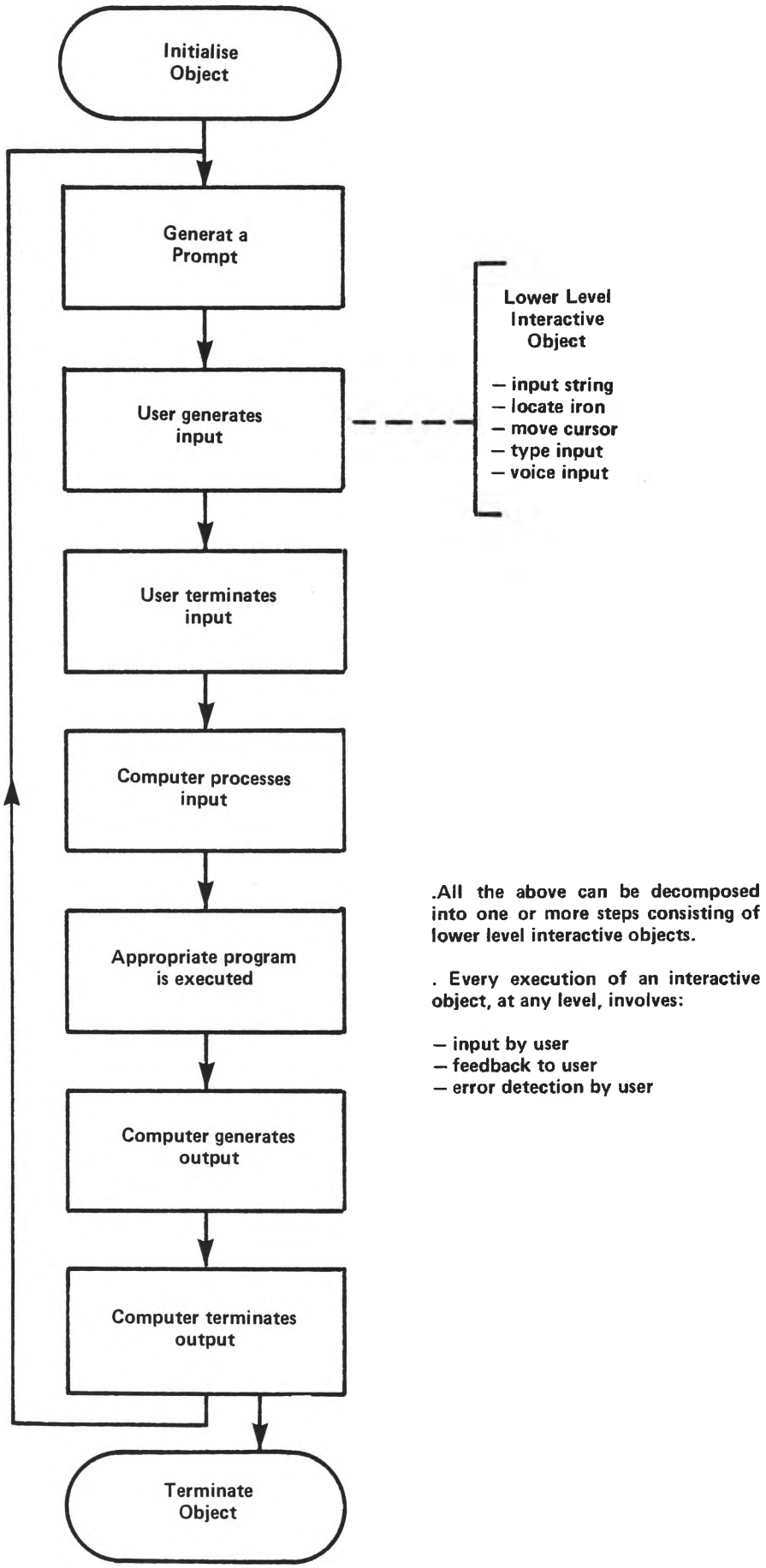


Figure 10.2 Interactive Object - a conceptual model of man-machine interaction.

effect user response time. W.K. English et al (1967) report on experiments to measure and compare response time for a variety of graphics input devices. E.B. Montgomery (1982) discusses methods of improving keyboard input, particularly wiping keys as an alternative to pressing keys.

Using the concepts of the formulation of performance measurement an interactive object (figure 10.2) has been developed. The object is a conceptual model of man-machine interaction consisting of a sequence of modules. The user-generates-input module is itself a lower-level interactive-object. Thus, our interactive object fits the conceptual model upon which the formulation of performance measurement is based, and hence all the measures are defined.

Every execution of an interactive object, at any level, involves: input by the user, feedback to the user, error detection by the user, and the generation of a new input by the user. The time taken by the user to think about a new input depends upon the nature of the task, whether it is procedural or problematic. Instrumentation of the object is the same as for any other object, but user activity can only be measured indirectly through the software. Normally we do not wire the user up to the measuring tool. The evaluation of the measured data involves all the behavioural issues, and hence requires a method of measuring user characteristics. This area requires a great deal more research, particularly in the area of the psychology of human-computer interaction (Card et al 1983).

10.3. Computer Networks

A network of computers consists of two or more computers linked together, while a computer network is either a network of computers or a set of terminals connected to one or more computers (Cole 1971, quoted by Morgan 1975). To effectively manage a computer system network, it is essential to monitor its behaviour as it executes a set of programs and responds to its environment.

Mendicino and Sutherland (1973) identify a number of measures for networks of terminals in their discussion of the Octopus Computer Network at the Lawrence Livermore Laboratory.

Response Time
- network delay
- host delay

Interactive throughput time

Interaction rate

Throughput rate

Availability

Reliability

Line utilization

Front-end utilization

Error rate

Message rate
- transmitted messages
- received messages

Turnround time

Quality of Messages

Transaction rate

Terminal utilization

Control unit utilization.

Table 10.2 Terminal Network Performance Parameters (Terplan 1981)

They measured the number of users logged on at regular time intervals and the volume of message traffic to and from the teletypes. Message traffic was divided into four categories: messages to the system, messages from the system, messages to programs, and messages from programs. More recent work at Lawrence Livermore (Brice and Alexander 1982) has involved an attempt to solve the problems of performance measurement on a large network of dissimilar computers (over 40 computers, 5 vendors, 7 operating systems), by concentrating all performance evaluation functions in one network node.

A survey of currently available performance monitors for terminal networks (Terplan 1981) includes a table of performance parameters for these networks (table 10.2). All of these parameters, except quality and possibility reliability, can be measured by carefully instrumenting the system in accordance with the methodology described in this thesis. The effective management of networks (Terplan 1982) requires performance measurement of all principal components in the network, preferably, with collected data being analysed, and displayed, at a centralised network-control-center.

Networks used to tightly couple processors together to form parallel processors have been discussed in the previous chapter. In this section I wish to concentrate on the areas of networking generally known as wide-area networks, and local-area networks.

A computer network monitoring system (CNMS) was developed at the University of Waterloo to measure **wide-area networks** (Morgan et al 1975, Buck and Hrynyk 1978). Although it was only used to measure a small network, it enabled the researchers to study a number of aspects of network monitoring. The papers give far more detail about the tool than the actual measures made. The tool consists of a set of hybrid monitors, each located at a remote network node, all communicating with and controlled by a central network monitor. The central system could place simulated loads on the network, control the remote monitors, receive data from the remote monitors and analyse the performance of the network.

The hybrid monitoring tools are identical in principle to those used to monitor computers. The reasons given for monitoring a network are the same as the reasons for monitoring a

computer. However, a problem exists in monitoring networks which does not occur in monitoring monoprocessors. The large physical distance between network nodes makes determining the order in which nearly simultaneous events occur difficult. In a multiprocessor a master synchronising clock is used, but in a network the time delays in sending the synchronisation signal from the control unit to a remote monitor may be significant.

The monitors are event driven monitors, where an event is a change in the system state. Two histograms were found to be useful for understanding the behaviour of the network: system state versus time in each state, and system state transitions versus the number of such transitions. A second problem in monitoring a wide-area network is getting the data from the remote monitor to the central controlling unit. Data reduction mechanisms must be used to reduce the data to manageable quantities.

Silvester and Kleinrock (1983) have analysed the performance of various configurations of ALOHA networks using models. The ALOHA network is a broadcast network connecting the island campuses of the University of Hawaii, based upon a transmit when ready, collision detection and random back off technique. Analysis of this system has shown the maximum that the channel can be utilized is 18 per cent of the channel band width. Modifying the scheme to force transmission to commence at the beginning of slots (time divisions of length equal to a packet transmission time) doubles the capacity of 36 per cent.

Measures of interest to Silvester and Kleinrock were:

- the number of successful transmissions per slot for any node in the network,
- the average path length, in terms of node-to-node transmissions, that messages traversed through the network, and
- network throughput.

Kleinrock and Opderbeck (1977) carried out a series of experiments to determine the speed at which large files could transfer through the ARPANET. They measured the time taken as a function of the number of hops (transmission from one node to the next) and calculated the throughput as a function of the number of hops. These measures gave them some feel for the

delays that occur at the nodes in the network. Investigation of occasional long network delays led to the discovery of extensive looping in the network. The dynamic routing algorithm could get into a situation where packets were tossed back and forth between neighbouring nodes many times, and thus, were delayed until the adaptive routing procedure corrected the anomaly.

Local area networks have a different set of characteristics to wide-area networks. The routing problem is removed because there is only one common transmission medium. Local area networks are characterised by the way they handle contention for the medium. Some use CSMA/CD (Carrier Sense Multiple Access/Collision Detect) technology similar to ALOHA net, the best known of these is Ethernet. Others obtain the network before transmission, for example token busses and token rings where once a station acquires a token it can transmit without fear of collision. Another method is the slotted ring technique used in the Cambridge Ring, where a fixed number of slots circulates around the network and a station can acquire the next empty one. Each network topology has its own advantages and disadvantages.

Measurement of these networks can be in three different areas. Measurement of **network performance** involves: measurement of the time to acquire the network, the delay in waiting for the network, the time to send messages over the network, and the degradation of the network under heavy load (Metcalfe and Boggs 1976 discuss the performance of Ethernet under load). The second area of network measurement involves measuring the **response time of acknowledgements**. Two types of acknowledge exist in a network: a low-level acknowledge to say that the message got there OK, and a higher-level answer to the message. These measures measure the computer at the other end as well as the network.

The third area of network measurement is the measurement of the **cost of the protocol handlers**, and the impact of network communications upon the computer using the network. Nelson (1981) discusses the performance of programs handling a remote procedure call mechanism over an Ethernet, and various ways of improving performance. Belanger et al (1981) discuss the performance of two experimental protocols for ZNET (a low cost Ethernet-like network). Their measurements indicate that making higher-level software responsible for reliable

message delivery greatly increases response time.

The above discussion is not intended to give a comprehensive history of network measurement, but rather to introduce the various measurement situations and to give a feel for the problems involved. Much more effort has gone into modeling computer networks than into measuring them. Donald DuBois (1982) describes a hierarchical modeling system, consisting of both analytical and simulation models, for use in the design of networks. Kleinrock's open queueing model for computer networks is used, and the following performance measures are derived: expected message delay at a node, utilization of the network, average queueing time at a node, and average number of messages at a node.

As with the other applications we have looked at in this chapter, if we can describe networks in terms of the conceptual model of an object then the measures are defined. We appear to have several network objects, one for each topology. However, some generalisations can be made.

If the network is looked at from the point of view of a program using the network then communication over the network is just a module in our program object. The network object consists of: the network driver, which formats the message, acquires the network, and transmits the message; the destination computer which answers the message; and the network driver which receives the answer and passes it to the program using the network. This type of network object is included in the class of normal monoprocessor objects. It can be used to measure the time taken to transmit messages over the network, the message-received-OK acknowledge-time, and the answer response time. However, it gives very little feel for the behaviour of the network as a whole.

Our conceptual model of a computer involves the flow of control of the program counter. If we think of a network in terms of the flow of control of messages, then we have a conceptual model of a network very similar to our conceptual model of a multiprocessor. A network object is a set of node modules connected by a set of transmission modules. The path of a message through the network is a sequence of modules. The time taken by the message to travel

through the network is the execution time of the object.

An event trace containing sufficient information to measure the network can be obtained if the arrival and departure of messages from nodes in the network are considered to be events. Stimulus information associated with each event is a message identifier, or in the case of a network where fixed path virtual links are established, the link identifier. Other stimulus information includes the number of messages queued at the node waiting for retransmission.

Measurement based upon the first network object defines network parameters from the point of view of a process using the network. Measurement based upon the second network object defines network parameters from the point of view of network loading and network performance. By combining these measurements a comprehensive picture of the flow of messages through a network, and the impact of these messages upon the sender and receiver can be obtained.

11. Conclusion

> *When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of a Science.*
>
> *Lord Kelvin (Dunham and Krusi 1983)*

In this dissertation, I have sought to codify the results of the last twenty-five years of research in performance measurement into a unified body of knowledge. As far as I can tell, I have included all the significant advances, and credited those who made them. I have concentrated on measurement, and I have only discussed performance evaluation, performance modelling and performance improvement at the points where they are impacted by measurement.

Unification of the field involved two interlocking processes: the development of a general formulation of performance measurement; and the study of measurement tools, techniques, and experiments in the context of the formulation. Most of the ideas presented in this dissertation are not new; the new element is the unification of the field based upon a formulation of performance measurement.

A computer program is an implementation of a mathematical object which can be measured. The dynamic performance of the object during execution can be measured, and the measures can be described mathematically. The object being measured can be as small as a single instruction or as large as a complete system. An object at one level can be decomposed into a set of objects at a lower level. A full decomposition of a computer system is given both for monoprocessors and multiprocessors.

One consequence of the hierarchical decomposition of the object is that a set of measures has been defined which applies to all levels of the object hierarchy. The conceptual model of an object during execution is a sequence of modules (lower level objects). The extent of an object is defined in terms of this conceptual model.

The object is instrumented to produce an event trace consisting of module start tuples, and stimulus information triples. From this event trace an object record is generated. Other measures are calculated from the object record. All measures can be displayed graphically for easy human analysis.

The formulation of performance measurement provides an overall context in which performance measurement experiments can be conducted. A methodology for performance measurement has been developed, and the integration of performance measurement tools into a system at design time discussed. A monitor chip for use with microprocessors, which will reduce the interference of the software probes in a hybrid tool, has been proposed.

A historical survey of measurement tools and techniques is included in the dissertation. A philosophy of hybridization has been developed from the performance measurement formulation, and a hybrid tool designed. A subset of this tool has been implemented and a number of measurement studies included. The tool is built from off-the-shelf components: a logic-state analyser, a personal computer, a communications link, a simple hardware probe, and software probes. The technology is now available to build a powerful, low-cost, general-purpose performance measurement tool. On the basis of the formulation of performance measurement, the characteristics of such a tool have been defined.

In the past, in the absence of a formulation of performance measurement, integration of performance measurement instruments into a system has been difficult because of the problem of specifying what to measure on a system whose design has not been stabilized. In response, a number of approaches were taken: no instrumentation was included, a general-purpose tool was added later, or a shot-gun approach to instrumentation was taken. In the latter, a lot of tools were included in the hope that nothing would be missed. Instrumentation on the basis of a general formulation of performance measurement can be tailored to the needs of the system, and at the same time will be general enough to get at all variables of interest, because we now understand performance measurement.

One topic in the area of performance measurement that has concerned a number of researchers is privacy and security. The tools discussed in this dissertation are able to get past the best security system, and hence could be used to invade privacy. Access to performance measurement tools integrated into a system must be carefully controlled. Performance measurement tools in themselves are neutral. It is the people who use them who choose to use them for good or evil. We each have a moral responsibility to our fellow man, and before God, to use these tools for ethical purposes.

A number of ways of validating the formulation of performance measurement have been proposed. A large part of this dissertation is dedicated to validating the formulation, and to demonstrating that it is general enough to codify the field of performance measurement into a unified body of knowledge. The research of others has been studied and compared to the performance measurement formulation. A number of experiments based upon the formulation have been conducted. Corollaries to the formulation have been hypothesised and tested. Only when several researchers have carried out independent experiments will the formulation of performance measurement become generally accepted as a framework within which to do performance measurement.

A number of further research projects, following on from this dissertation, have been mentioned. One is independent validation of the formulation of performance measurement. Another is extending the formulation to cover performance evaluation. Practical extensions of this research are the development of the general-purpose tool defined in this dissertation, and the integration of performance tools into new processors, via the development of monitor chips.

12. Bibliography

Of making many books there is no end,
and much study wearies the body.

Ecclesiastes 12:12b

12.1. Bibliographies

Agajanian A.H. (1975), A Bibliography on System Performance Evaluation, IEEE Computer, Vol 8, No 11, November, pp 63-74.

Miller E.F. (1972), Bibliography on Techniques of Computer Performance Analysis, IEEE Computer, Vol 5, No 5, September, pp 39-47.

Shaw M. (1981), Annotated Bibliography on Software Metrics, in Perlis A. et al (Eds), Software Metrics: An Analysis and Evaluation, MIT Press.

Tanik M.M. (1980), An Annotated Bibliography on Performance Evaluation, Performance Evaluation Review, ACM Sigmetrics, Vol 9, pp 24-30.

12.2. Journals and Special Journal Issues

Computer Performance, *Vince N.C.* (Ed), Butterworth Scientific Limited, P.O. Box 63, Westbury House, Bury St, Guildford, Surrey GU2 5BH, England.

EDP Performance Review, Applied Computer Research, Phoenix, Arizona.

Performance Evaluation, *Kobayashi H.* (Ed), North-Holland Publishing Company, P.O. Box 103, 1000 AC Amsterdam, The Netherlands.

Performance Evaluation Review, *Highland H.J.* (Ed) ACM Sigmetrics, 11W. 42nd St, New

York, NY 10036.

Agrawala A.K. and Herzog U. (Eds) (1983), IEEE Transactions on Computers, Vol C-32, No 1, January.

Ferrari D. (Ed) (1976), Program Behaviour, IEEE Computer, Vol 9, No 11, November.

Gillette D. (Ed) 1978, Unix Time-Sharing System, The Bell System Technical Journal, Vol 57, No 6, Part 2, July.

Graham G.S. (Ed) (1978), Computing Surveys, Vol 10, No 3, September.

Shemmer J.E. (Ed) (1972), IEEE Computer, Vol 5, No 4, July.

Shively R.R. (Ed) (1972), IEEE Computer, Vol 5 No 5, September.

Spragins J. (Ed) (1980), Analytical Queueing Models, IEEE Computer, Vol 13, No 4, April.

12.3. Books, Theses, Conference Digests, and Equipment Manuals

Apple II - IEEE Bus Interface (1979), Model IE-01-79, Kayin Peripherals, Baulkham Hills, N.S.W.

Apple Pascal (1980), Operating System Reference Manual, Apple Computer Inc., Cupertino, California.

Beilner H. and Gelenda E. (1977), Measuring Modelling and Evaluating Computer Systems, North-Holland.

Borovits I. and Neumann S. (1979), Computer Systems Performance Evaluation, Lexington Books, Massachusetts.

Bunyan C.J. (Ed) (1974), Computer Systems Measurement, Infotech State of the Art, Report 18.

Card S.K., Moran T.P., Newell A. (1983), The Psychology of Human-Computer Interaction, Lawrence Erlbaum, New Jersey.

Chandy K.M. and Reiser M. (Ed) (1977), Computer Performance, North-Holland.

Chang D.Y-C. and Lawrie D. (1982), Performance of Multiprocessor Systems with Space and Access Contention, Report No UIUCDCS-R-82-1085, Department of Computer Science, University of Illinois at Urbana-Champaign.

Cooper R. (1981), Introduction to Queueing Theory, Second Edition, North-Holland.

Courtois P.J. (1977), Decomposability, Queueing and Computer System Applications, ACM Monograph Series, Academic Press.

Cutland N.J. (1980), Computability - an introduction to recursive function theory, Cambridge University Press.

de Bakker J. (1980), Mathematical Theory of Program Correctness, Prentice Hall.

Dijkstra E.W. (1976), A Discipline of Programming, Prentice Hall.

Drummond M.E. (1973), Evaluation and Measurement Techniques for Digital Computer Systems, Prentice Hall.

Dumont D.N. (1978), Computer Performance Measurement Tools and Techniques, University of California, Santa Barbara, Ph.D., University Microfilms International No 7921039.

Ferrari D. (1978a), Computer Systems Performance Evaluation, Prentice Hall.

Ferrari D. (Ed) (1978b), Performance of Computer Installations, North-Holland.

Ferrari D. & Spadoni M. (Ed) (1981), Experimental Computer Performance Evaluation, North-Holland.

Ferrari D., Serazzi G., & Zeigner A. (1983), Measurement and Tuning of Computer Systems, Prentice Hall.

Fosdick L.D. (Ed) (1979), Performance Evaluation of Numerical Software, North-Holland.

Graham G.S. (1981), Computer Systems Performance Evaluation, Lecture Notes Advanced Management Research Symposium, Sydney.

Halstead M.H. (1977), Elements of Software Science, Elsevier, New York.

Hanson P.B. (1973), Operating Systems Principles, Prentice Hall.

Hewlett-Packard (1978a), Operating Guide 1610A Logic State Analyser, February.

Hewlett-Packard (1978b), Seminar in Problem-Solving Concepts for Computer Systems, Part III, Complex Bus and Software Analysis Introduction, October, pp 53-80.

Jenkins J.H. (1975), Assessment of the Control Structure of a Minicomputer using a hardware monitor, Ph.D. Thesis, University of California Santa Barbara.

Kobayoshi H. (1978), Modelling and Analysis: an Introduction to Performance evaluation Methodology, Addison Wesley.

Knuth D.E. (1968), The Art of Computer Programming, Vol 1: Fundamental Algorithms, Addison Wesley.

Kylstra F.J. (Ed) (1981), Performance '81, North-Holland.

Lavenberg S. (Ed) (1983), Computer Performance Modelling Handbook, Academic Press.

Lanciaux D. (Organizer) (1978), Second Colloque International Sur les Systemes D'Exploitation, IRIA - Rocquencourt, 2-4 Octobre, Le Chesnay, France.

Liguori F., Automatic Test Equipment, IEEE Press.

Lions J. (1977), A Commentary on the Unix Operating System, University of New South Wales.

Manna Z. (1974), Mathematical Theory of Computation, McGraw-Hill.

Marathe M.V. (1977), Performance Evaluation at the Hardware Architecture Level and the Operating System Kernel Design Level. Ph.D. Thesis, Computer Science Department, Carnegie Melon University, Pittsburg.

McKerrow P.J. (1978), Control of Coating Mass on a Continuous Hot Dip Galvanizing Line, Master of Engineering Thesis, The University of Wollongong.

Nelson B.J. (1981), Remote Procedure Call, XEROX, Palo Alto Research Center, Report No CSL-81-9.

Perlis A.J., Sayward F.G., and Shaw M. (Eds) (1981), Software Metrics, MIT Press.

Sauer C.H. and Chandy K.M. (1981), Computer Systems Performance Modelling, Prentice Hall.

Sprin J.R. (1977), Program Behaviour: Models and Measurements, Elsevier.

Svobodova L. (1976a), Computer Performance Measurement and Evaluation Methods: Analysis and Application, Elsevier.

Taub A.H. (Ed) (1963), von Newmann's Collected Works, Vol 5, Pergamon.

Wand, M. (1980), Induction, Recursion and Programming, Elsevier, North-Holland.

Walpole R.E. and Myers R.H. (1978), Probabilty and Statistics for Engineers and Scientists, Collier MacMillan.

White C.H. (Ed) (1977), Systems Tuning, Infotech State of the Art Report.

12.4. Papers

Adrion W.R. (1982), Validation, Verification, and Testing of Computer Software, Computing Surveys Vol 14, No 2, June, pp 159-192.

Allan R. (1977), Logic Analyzers, IEEE Spectrum, Vol 14, No 8, August.

Amiot L.W. et al (1972), Evaluating a Remote Batch Processing System, Computer, September/October, pp 24-29.

Apple C.T. (1965), The program monitor - A device for program performance measurement, Proc of the ACM 20th National Conference, pp 65-75.

Arndt F.R. and Oliver G.N. (1971), Hardware Monitoring of Real-Time Computer System Performance, Digest of 1971, IEEE International Computer Society Conf., pp 123-125.

Aschenbrenner R.A., Amiot L. and Natarajan (1971), The neurotron monitor system, AFIPS, FJCC, Vol. 39, pp 31-37.

Balzer R.M. (1969), Exdams - Extended Debugging and Monitoring System, AFIPS SJCC, Vol 34, pp 567-580.

Barnes G.H. et al (1968), The Illiac IV Computer, IEEE Transactions on Computers, Vol. C-17, August, pp 746-757.

Bauer M.J. and McCredie J.W. (1973), AMS: A Software Monitor for Performance Evaluation and System Control, Proc First Annual SICME Symposium on Measurement and Evaluation, February, pp 147-160.

Bell T.E., Boehm B.W., and Watson R.A. (1972), Framework and Initial Phases for Computer Performance Improvement, AFIPS, No 41, pp 1141-1154.

Bergerol C. (1981), Distributed Performance Monitoring System, Computer Performance, IPC

Science and Technology Press Limited, Vol 2, No 1, pp 5-12.

Belanger P. et al (1981), Performance Measurements of a Local Area Network, in A. West and P. Janson (eds), Local Networks for Computer Communications, North-Holland, pp 181-189.

Bisiani R., Mauersberg H., and Reddy R. (1983), Task-Oriented Architectures, Proc of the IEEE, Vol 71, No 7, pp 885-898.

Blake R. (1980), XRAY: Instrumentation for Multiple Computers, ACM Sigmetrics, Performance Evaluation Review, Vol 9, part 2, pp 11-25.

Boulaye G. et al (1977), A Computer Measurement and Control System, Proc 3rd International Symposium, Measuring, Modelling and Evaluating Computer Systems, October, North Holland.

Bourret P. and Cros P. (1980), Presentation and Correction of Errors in Operating System Measurements, IEEE Transactions on Software Engineering, Vol 6, No 4, July, pp 395-398.

Boyle B. (1984), Software Performance Evaluation, Byte, February, pp 175-188.

Brampton J. (1982), State Analyzers move from lab to production area, Electronic Design, May 13, pp 167-176.

Brice R. & Alexander W. (1982), A Network Performance Analyst's Workbench, Performance Evaluation Review, Vol 11 No 1, pp 138-146.

Browne J.C. and Shaw Mary (1981), Toward a Scientific Basis for Software Evaluation, in Perlis A. et al (Eds), Software Metrics: An Analysis and Evaluation, MIT Press, pp 19-41.

Buck D.L. and Hrynyk D.M. (1978), Software Architecture for a Computer Network Monitoring System, in Ferrari D. (Ed), Performance of Computer Installations, pp 269-287.

Bussell B. and Koster R.A. (1970), Instrumenting Computer Systems and their Programs, AFIPS FJCC, Vol 37, pp 525-534.

Buzen J.P. (1973), Computational Algorithms for Closed Queueing Networks with Exponential Servers, Comm ACM, Vol 16 No 9, pp 527-531.

Buzen J.P. (1976), Fundamental Operational Laws of Computer System Performance, Acta Informatica 7, 2, Springer Verlag, pp 167-182.

Cabrera L.F. (1981), Benchmarking Unix - A Comparative Study, Experimental Computer Performance Evaluation, North-Holland.

Calingaert P. (1967), System Performance Evaluation: Survey and Appraisal, Comm ACM, Vol 10, No 1, pp 12-18.

Campbell D.J. and Heffner W.J. (1968), Measurement and Analysis of Large Operating Systems during System Development, AFIPS, FJCC, Vol 33, pp 903-914.

Cantrell H.N. and Ellison A.L. (1968), Multiprogramming System Performance Measurement and Analysis, Proc AFIPS, SJCC, Vol 32, pp 213-221.

Carey T. (1982), User Differences in Interface Design, IEEE Computer, Vol 15, No 11, November, pp 14-20.

Carlson G. (1971), A User's View of Hardware Performance Monitors or How to Get More Computer for Your Dollar, IFIP Congress 1971, pp 128-132.

Carlson G. (1976), A Guide to the Use of Hardware Monitors -- Part 2. EDP Performance Review, Vol 4, No 10, October.

Carlson G. (1977), Hardware Monitoring for System Tuning, Infotech Reports System Tuning, pp 255-274.

Carver Hill J. (1974), The State of logic Analyzers, IEEE Spectrum, Vol 11, No 12, December, pp 63-70.

Chandy K.M. and Sauer C.H. (1980), Approximate Solution of Queueing Models, IEEE Computer, Vol 13, No 4, April, pp 25-32.

Charlton C.C. and Mander K.C. (1983), Tools and Techniques for Teaching Microprocessor Software Development, Software Practice and Experience, Vol 13, No 10, October, pp 909-920.

Cheriton D.R. et al (1979), Thoth, a Portable Real-Time Operating System, Communications of the ACM, Vol 22, No 2, pp 105-115.

Christensen K., Fitsos G.P. and Smith C.P. (1981), A Perspective on Software Science, IBM Syst. J, Vol 20 No 4, pp 372-387.

Cole, G.D. (1971), Computer network measurements - techniques and experiments, University of California, Los Angeles, NTIS, Report AD-739-344, October.

Collins J.P. (1976), Performance Improvement of the CP-V Loader through use of the Adam Hardware Monitor, Performance Evaluation Review, Vol 5, No 2, April, pp 63-68.

Comerford R.W. (1981), Measurement-Computer Era Arrives, Electronics, September 8, pp 96-100.

Constantine L.L. (1968), Integral Hardware/Software Design - Part 7 - Formalization of Computer Power, Modern Data, November, pp 58-66.

Corson D. (1983), Logic Tool Analyses Software Performance, Electronic Design, January 20, pp 117-126.

Coulter N.S. (1983), Software Science and Cognitive Psychology, IEEE Transactions on Software Engineering, Vol SE9, No 2, March, pp 166-171.

Coutant C.A. et al (1983), Measuring the Performance and Behaviour of Icon Programs, IEEE Transactions on Software Engineering, Vol SE9, No1, Jan. pp 93-103.

Cureton H.O. (1972), A Philosophy to System Measurement, AFIPS, Vol 41, part II, FJCC, pp 965-969.

Curtis B. (1980), Measurement and Experimentation in Software Engineering, Proc of the IEEE, Vol 68, No 9, September, Special Issue on Software Engineering, pp 1144-1157.

Deese D.R. (1974), A Computer Design for Measurement - The monitor register concept, Computer Performance Evaluation, U.S. Department of Commerce, N.B.S. Special publication 401, pp 63-72.

Deniston W.R. (1969), SIPE: A TSS/360 Software Measurement Technique, Proc 24th ACM National Conference, pp 229-245.

Denning P.J. and Buzen J.P. (1978), The Operational Analysis of Queueing Network Models, Computing Surveys, Volume 10, Number 3, September, pp 255-261.

Denning P.J. (1978), Working Sets Then and Now, Proc of the 2nd Colloque International sur les Systemes D'Exploitation, IRIA, Rocquencourt, October.

Denning P.J. (1981), Performance Analysis: Experimental Computer Science at Its Best, Communications of the ACM, Vol 24, No 11, November, pp 725-727.

Denny W.M. (1977), The Burroughs B1800 Microprogrammed Measurement System: A Hybrid Hardware/Software Approach, Proc of the 10th Annual Microprogramming Workshop called MICRO 10, ACM, New York.

De Prycker M. (1982), A Performance Analysis of the Implementation of Addressing Methods in Block-Structured Languages, IEEE Transactions on Computers, Vol C-31, No 2, February, pp 155-163.

De Prycker M. (1982), On the Development of a Measurement System for High Level language program Statistics, IEEE Transactions on Computers, Vol C-31, No 9, September, pp 883-891

Deutch P. and Grant C.A. (1971), A Flexible Measurement Tool for Software Systems, Information Processing 71, Proc IFIP Congress 71, North Holland.

DuBois D.F. (1982), A Hierarchical Modelling System for Computer Networks, Performance Evaluation Review, ACM Sigmetrics, Vol 1, No 1, pp 147-155.

Dunham J.R. and Kruesi E. (1983), The Measurement Task Area, IEEE Computer, Special Issue on the DoD STARS Program, Vol 16, No 11, pp 47-54.

English W.R., Engelbart D.C. and Berman M.L. (1967), Display - Selection Techniques for Text Manipulation, IEEE Transactions on Human Factors in electronics, Vol 8 No 1, pp 5-20.

Estrin G., Hopkins D., Coggan B. and Crocker S.D. (1967), SNUPER COMPUTER - A Computer in Instrumentation Automation, AFIPS, SJCC, Vol 30 (SJCC), pp 645-656.

Estrin G., Muntz R.R. and Uzgalis R.C., (1972), Modelling, measurement and computer power, AFIPS, SJCC, Vol 40, pp 725-738.

Estrin G. (1974), Measurable Computer Systems, Infotech Reports, Vol 18, pp 285-299.

Fairclough D.A. (1982), A Unique Microprocessor Instruction Set, IEEE Micro, Vol 2, No 2, pp 8-18.

Febish G.J. (1981), Experimental Software Physics, in Ferrari D. (Ed), Experimental Computer Performance Evaluation, North-Holland, pp 33-56.

Ferrari D. (1972), Workload characterisation and selection in computer performance measurement, IEEE Computer, Vol 5, No 4, July, pp 18-24.

Ferrari D. (1973), Architecture and instrumentation in a modular interactive system, IEEE Computer, Vol 6, No 11, pp 25-29.

Ferrari D. and Liu M. (1975), A General-Purpose software measurement tool,

Software - Practice and Experience, Vol 5, No 2, 181-182.

Ferrari D. (1981a), Characterization and Reproduction of the Referencing Dynamics of Programs, Performance 81, North-Holland, pp 363-372.

Ferrari D. and Minetti V. (1981b), A Hybrid Measurement Tool for Minicomputers, Experimental Computer Performance Evaluation, North-Holland, pp 217-233.

Fitzsimmons A. and Love T. (1978), A Review and Evaluation of Software Science, Computing Surveys, Vol 10, No 1, March 78, pp 3-18.

Flynn M. (1972), Some Computer Organizations and their Effectiveness, IEEE Transactions on Computers, C-21, Vol 9, September, pp 948-960.

Ford P. (1981), Simulation in Computer Performance Evaluation, Computer Performance, IPC Business Press, Vol 2, No 4, pp 186-191.

Foster C.C. et al (1971), Measures of Op-Code Utilization, IEEE Transactions on Computers, Vol C-20, No 5, May, pp 582-584.

Franta W.R. et al (1982), Issues and Approaches to Distributed Testbed Instrumentation, IEEE Computer, Vol 15, No 10, October, pp 71-81.

Freibergs I.F. (1968), The Dynamic Behaviour of Programs, AFIPS, FJCC, Vol 33, pp 1163-1167.

Fromm H. et al (1983), Experiences with Performance Measurement and Modelling of a Processor Array, IEEE Transactions on Computers, Vol C32, No 1, January, pp 15-31.

Fryer R.E. (1973), The Memory Bus Monitor - A New Device for Developing Real-Time Systems, AFIPS, NCC, Vol 42, pp 75-79.

Fuller S.H. et al (1973), The Instrumentation of C.mmp - A Multi-Mini-Processor, Proc

COMPCON 73, Seventh Annual IEEE Computer Society International Conference, pp 173-176.

Geck A. (1979), Performance Improvement by Feedback control of the Operating System, Performance of Computer Systems, North-Holland, pp 459-471.

Gehringer E.F. et al (1982), The CM* Testbed, IEEE Computer, Vol 15, No 10, October, pp 40-53.

Gerla M. and Kleinrock L., On the Topological Design of Distributed Computer Networks, IEEE Transactions on Communications Vol COM-25, No 1, January, pp 48-60.

Gibson J.C. (1970), The Gibson mix, IBM Tech Report, TR00 2043, June.

Glass R.L. (1980), Real Time: The 'Lost World' of Software Debugging and Testing. Communications of the ACM, Vol 23, No 5, May, pp 264-271.

Goodman C.D. (1963), Getting Multichannel Analyser Data in and out of the IBM 7090 for processing, Oakridge National Laboratory, Tech report.

Goodman A.F. (1972), Measurement of Computer Systems - An Introduction, AFIPS, FJCC, No 41, pp 669-680.

Gotlieb C.C. (1973), Performance Measurement, in Bauer F.L. (ED), Advanced Course on Software Engineering, Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, pp 464-491.

Graham R.M. (1973), Performance Prediction, in Bauer F.L. (Ed), Advanced Course on Software Engineering, Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, pp 395-463.

Graham S.L. et al (1983), An Execution Profiler for Modular Programs,
Software - Practice and Experience, Vol 13, pp 671-685.

Grishman R. (1971), "Criteria for a Debugging Language, in Rustin R. (Ed), Debugging Techniques in Large Systems, 1st Courant Computer Science Symposium, Prentice Hall, pp 57-76.

Grochow J.M. (1969), Real-time Graphic Display of Time-Sharing System Operating Characteristics, AFIPS, JFCC, Vol 35, pp 379-386.

Grosch H.R.J. (1953), High Speed Arithmetic - The digital computers as a research tool, J Optical Society of America, Vol 43, No 4, pp 306-310.

Guteri F. (1982), Design case history: Biomations Logic Analyser, IEEE Spectrum, Vol 19 No 5, May

Hamming R.W. (1980), Error Detecting and Error Correcting Codes, Bell System Technical Journal, Vol 26, No 2, pp 147-160.

Haynes L.S. et al (1982), A Survey of Highly Parallel Computing, IEEE Computer, Vol 15, No 1, January, pp 9-24.

Hercksen U., Klar R., Kleinoder W., Kneibl F. (1982), Measuring Simultaneous Events in a Multiprocessor System, Performance Evaluation Review, Vol 11, No 4, pp 77-88.

Hellerman L. (1972), A Measure for Computational Work, IEEE Transactions on Computers, Vol C-21, No 5, May 1972, pp 439-446.

Hempy H. (1977), IBM 3850 Mass Storage System, Performance Evaluation Using a Channel Monitor, in Chandy K.M. and Reiser M. (Eds), Computer Performance, North-Holland, pp 177-196.

Herbst E.H., Metropolis N. and Wells K.B. (1955), Analysis of problem codes on the Maniac, Maths tables and other aids to Computation, Vol 9 No 49, pp 14-20.

Hoare C.A.R. (1969), An Axiomatic Basis for Computer Programming, Communications of the ACM, Vol 12, No 10, pp 576-583.

Holdsworth D. (1983), A System for Analysing Ada Programs at Run-Time, Software - Practice and Experience, Vol 13 No 5, pp 407-421.

Holtwick G.W. (1971), Designing a Commercial Performance Measurement System, ACM Sigops Workshop on System Performance Evaluation, Cambridge, Massachusetts, pp 29-58.

Howard P.C. (1974), Update on Hardware Monitors, EDP Performance Review, Vol 2, No 10, October.

Huang J.C. (1980), Instrumenting Programs for Symbolic-Trace Generation, IEEE Computer, Vol 13, No 12, December, pp 17-23.

Hughes J. and Cronshaw D. (1973), On Using a Hardware Monitor as an Intelligent Peripheral, Performance Evaluation Review, 2, 4, pp 3-19, December.

Hughes J.H. (1980), Diamond - A digital Analyser and Monitoring Device, ACM Sigmetrics, pp 27-34.

Hughes P.H. and Moe G. (1973), A Structural Approach to Computer Performance Analysis, AFIPS, Vol 42, pp 109-119.

Ingalls D. (1972), The Execution time profile as a Measurement Tool, in Rustin R. (Ed), Design and Optimization of Compilers, Prentice Hall, pp 129-136.

Johnston S. (1980), Development of a software monitor, Computer Performance, Vol 1, No 2, pp. 61-80.

Johnson S.C. and Ritchie D.M. (1978), UNIX time-sharing: Portability of C programs and the UNIX system, Bell System Tech. J. Vol 57, No 6, pp 2021-2048.

Kashtan D.L. (1980), UNIX and VMS Some Performance Comparisons, SRI International.

Kearns J.P. et al (1982), The Performance Evaluation of Control Implementations, IEEE Tran-

sactions on Software Engineering, Vol SE8, No 2, March, pp 89-96.

Keefe D.D. (1968), Hierarchical Control Programs for Systems Evaluations, IBM Systems Journal, Vol 7, No 2, pp 123-133.

Kernighan B.W. and Mashey J.R. (1981), The UNIX programming environment, IEEE Computer, Vol 24, No 4, pp 12-24.

King P.J.B., Mitrani I. (1982), Modelling the Cambridge Ring, Performance Evaluation Review, ACM Sigmetrics, Vol 11, No 4, pp 250-258.

Klar R. (1981), Hardware measurements and their applications on performance evaluations in a processor-array, Computing, Suppl 3, pp 65-88, Springer Verlag.

Kleinrock L. (1966), Sequential Processing Machines (S.P.M.) Analyzed With a Queueing Theory Model, Journal of the Association of Computing Machinery, Vol 13, No 2, pp 179-193.

Kleinrock L. and Opderbeck H. (1977), Throughput in the ARPANET - Protocols and Measurement, IEEE Transactions on Communications, Vol COM25, No 1, January, pp 95-104.

Knuth D.E. (1971), An Empirical Study of Fortran Programs, Software - Practice and Experience Vol 1, pp 105-133.

Knuth D.E. (1977), Algorithms, Scientific American, Vol 236, No 4, pp 63-80, April.

Kolence K.W. (1971), A Software View of Measurement Tools, Datamation, Vol 17, No 1, January, pp 32-38.

Kolence K.W. (1972), Software Physics and Computer Performance Measurements, Proc ACM National Conference, pp 1024-1040.

Kolence K.W. and Kiviat P.J. (1973), Software unit profiles and Kiviat figures, Performance Evaluation Rev, Vol 2, No 3, September, pp 2-12.

Kolence K.W. (1975), Software Physics, Datamation, June, pp 48-51.

Kuck D.J. (1973), Complexity of Sequential and Parallel Numerical Algorithms, Traub J.F. (Ed), Academic Press, New York.

Kuck D.J. (1977), A survey of parallel machine organisation and programming, ACM Computing Surveys, Vol 9, March, pp 29-59.

Kumar B. and Davidson E.S. (1980), Computer System Design Using a Hierarchical Approach to Performance Evaluation, Communications of the ACM, Vol 25, No 9, pp 511-521.

Larsen R.L., Agre J.R., and Agrawala A.K. (1981), A Comparative Evaluation of Local Area Communication Technology, Performance Evaluation Review, ACM Sigmetrics, Vol 10 No 2, pp 37-47.

Lazos C. and Yandle J.R., Improving CPU Utilisation in a Multiprogramming System, The Computer Journal, Vol 22, No 3, pp 203-205.

Lee K.F. and Smith A.J. (1984), Branch Prediction Strategies and Branch Target Buffer Design, IEEE Computer, Vol 17, No 1, pp 6-22.

Lerner E.J. (1982), Instrumentation, IEEE Spectrum, Vol 19, No 1, January.

Lonergan R. and Androxiani V. (1970), SUPERMON: A Software Monitor for Performance Evaluation, Technical Memo No 30, Stanford Computation Centre, California, January.

Lorentzen R. (1979), Troubleshooting Microprocessors with a Logic-Analyzer System, Computer Design, March 1979.

Lucas H.C. (1971), Performance Evaluation and Monitoring, Computing Surveys, Vol 3, No 3, September, pp 79-91.

Lynch W.C. (1972), Operating System Performance, Communications of the ACM, Vol 15, No

7, July 1972, pp 579-585.

MacEwen G. (1974), On Instrumentation Facilities in Programming Languages, IFIP Congress, North-Holland, pp 198-203.

McCarthy J. (1960), Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1, Communications of the ACM, pp 184-204.

McCarthy J. (1962), Towards a mathematical science of computation, Proc IFIP, North-Holland, pp 21-28.

McDaniel G. (1982), The Mesa Spy: An Interactive Tool for Performance Debugging, Performance Evaluation Review, ACM Sigmetrics, Vol 11, No 4, pp 68-76.

McKerrow P.J. (1983), Evaluation of Interrupt-Handling Routines with a Logic-State Analyser, Performance Evaluation, North-Holland, Vol 3, pp 277-288.

McKerrow P.J. (1984), Monitoring Program-Execution with a Hybrid Measurement-Tool, Australian Computer Science Communications, Vol 6, No 1, pp 22.1-22.10.

McQuillan J.M. et al (1978), A Review of the Development and Performance of the ARPANET Routing Algorithm, IEEE Transactions on Communications, Vol COM-26, No 12, December, pp 1802-1811.

Mahjoub A. and Ekanadham K. (1980), A Compile Time Approach to Response Time Computation, Computer Science Department, SUNY at Stony Brook, N.Y.

Mendicino S.F. and Sutherland G.F. (1973), Performance measurements in LLL Octopus Computer Network, IEEE Compcon 73, Digest of Papers, pp 109-112.

Metcalfe R.M. and Boggs D.R. (1976), Ethernet: Distributed Packet Switching for Local Computer Networks, Communications of the ACM, Vol 19, No 7, July, pp 395-404.

Miller L.A. and Thomas J.C. (1977), Behavioural issues in the use of interactive systems, International Journal of Man-Machine Studies, Vol 9, pp 509-536.

Miller R. (1978), UNIX - A portable operation system? Operating Systems Review, Vol 12 No 3, pp 32-37.

Miller R.B. (1968), Response Times in Computer Conversation Transactions, AFIPS, FJCC, Vol 33, pp 267-277.

Montgomery E.B. (1982), Bringing Manual Input into the 20th Century: New Keyboard Concepts, IEEE Computer, Vol 15, No 3, pp 11-18.

Morgan D.E. et al (1974), A Performance Measurement System for Computer Networks, Information Processing 74, North-Holland.

Morgan D.E., Banks W., Goodspeed D.P., Kolanko R. (1975), A Computer Network Monitoring System, IEEE Transactions on Software Engineering, Vol SE-1, No 3, September, pp 299-311.

Morris M.F. (1976), Kolence: true or false?, Computerworld, October 18, page 1.

Mosher D.A. (1980), Unix Performance An Intraspection, Ampex Corporation, Redwood City, California.

Murphy R.W. (1969), The System Logic and Usage Recorder, AFIPS, FJCC, Vol 35, pp 219-229.

Nievergelt J. (1982), Errors in Dialog Design and How to Avoid Them, in Nievergelt J. et al (Eds), Document Preparation Systems - A Collection of Survey Articles, North-Holland, pp 1-10.

Nemeth A.G. and Rovner P.D. (1971), User Program Measurement in a Time-Shared Environment, Communications of the ACM, Vol 14, No 10, pp 661-666, October.

Nutt G.J. (1975), Tutorial: Computer System Monitors, IEEE Computer, Vol 8, No 11, November, pp 51-61.

Parkinson D. and Liddell H.M. (1983), The Measurement of Performance on a Highly Parallel System, IEEE Transactions on Computers, Vol C32, No 1, January, pp 32-37.

Patterson D.A. and Piepho R.S. (1982), Assessing RISCs in High-Level Language Support, IEEE Micro, Vol 2, No 4, pp 9-19.

Patterson D.A. and Sequin C.H. (1982), A VLSI Risc, IEEE Computer, Vol 15, No 9, pp 8-18.

Pearson S.W. & Bailey J.E. (1980), Measurement of Computer User Satisfaction, Performance Evaluation Review, ACM Sigmetrics, pp 59-68.

Penny J.P. and Sheedy C.R. (1980), Measurement of Response Time Performance in Small Time-Sharing Systems, The Australian Computer Journal, Vol 12, No 1, February.

Peterson T.G. (1974), A Comparison of Software and Hardware Monitors, Performance Evaluation Review, Vol 3, No 2, June, pp 2-5.

Pfaff G., Kuhlmann H. and Hanusa H. (1982), Constructing User Interfaces Based on Logical Input Devices, IEEE Computer, Vol 15, No 11, November, pp 62-68.

Plattner B. and Nievergelt J. (1981), Monitoring Program Execution: A Survey, IEEE Computer, November, Vol 14 No 11, pp 76-93.

Pomeroy J.W. (1972), A guide to programming Tools and Techniques, IBM Sys Journal, Vol 11, No 3, pp 234-254.

Price T.G. (1976), A Comparison of Queueing Network Models and Measurements of a Multiprogrammed Computer System, SIGMETRICS Performance Evaluation Review, Vol 5, No 4, pp 39-62.

Prichard E.L. (1976), Logic of software physics gives guide to change, Computerworld, October 18, pp 19-20.

Rafii A. (1981), Structure and Application of a Measurement Tool-Sampler/3000, Performance Evaluation Review, ACM Sigmetrics, Vol 10, No 3, pp 110-120.

Rajulu R.G. and Rajaraman (1982), Execution-Time Analysis of Process Control Algorithms on Microprocessors, IEEE Transactions on Industrial Electronics, Vol IE-29, No 4, November pp 312-319.

Reinfelds J. (1983), High-level traces, Software Engineering lectures, University of Wollongong.

Rhodes F.H.T. (1965), Christianity in a Mechanistic Universe, in MacKay D.M. (Ed), Christianity in a Mechanistic Universe and other Essays, Inter-Varsity Press, London, pp 11-48.

Rodriguez-Rosell J. (1976), Empirical Data Reference Behaviour in Data Base Systems, IEEE Computer, Vol 9, No 11, November, pp 9-13.

Roek D.J. and Emerson W.C. (1969), A Hardware Instrumentation Approach to Evaluation of a Large Scale System, ACM National Conference Proc.

Rose C.A. (1977), A Calibration - Prediction Technique for Estimating Computer Performance, AFIPS, NCC, Vol 46, pp 813-181.

Rose C.A. (1978), A Measurement Procedure for Queueing Network Models of Computer Systems, Computing Surveys, Vol 10, No 3, September.

Rosen S. (1968), Hardware Design Reflecting Software Requirements, AFIPS, FJCC, Vol 33, Part 2.

Rozwadowski R.T. (1973), A measure for the Quantity of Computation, Proc. First ACM/SIGME Symposium on Measurement and Evaluation, ACM New York, pp 100-111.

Ruud R.J. (1972), The CPM-X, A Systems Approach to Performance Measurement, Proc FJCC, Vol 41, Part II, pp 949-957.

Saal H.J. and Shustek L.J. (1972), Microprogrammed Implementation of Computer Measurement Techniques, Proc ACM 5th Annual Workshop on Microprogramming, September, pp 42-50.

Sakman H., Erikson W.J. and Grant E.E. (1968), Exploratory Experimental Studies Comparing Online and Offline Programming performance, Comm ACM, Vol 11, January.

Saltzer J.H. and Gintell J.W. (1970), The Instrumentation of Multics, Comm of the ACM, Vol 13, No 8, pp 495-500.

Saunders R.S.J. (1981), The PDSP - A performance Measurement tool for a Packet-Switching Network, Performance 81, North-Holland, pp 531-545.

Schach S.R. (1982), A Unified Theory for Software Production, Software - Practice and Experience, Vol 12, pp 683-689.

Schulman F.D. (1967), Hardware Measurement device for IBM system/360 time sharing evaluation, Proc of the ACM National Meeting.

Sebastian P.R. (1974), HEMI - Hybrid Events Monitoring Instrument, 2nd Sigmetrics Symposium on Measurement and Evaluation Proc, Montreal, pp 127-139.

Sedgewick R. et al (1970), SPY--A program to Monitor OS/360, Proc AFIPS FJCC, Vol 37, pp 119-128.

Segall Z. et al (1983), An Integrated Instrumentation Environment for Multiprocessors, IEEE Transactions on Computers, Vol C32, No 1, January, pp 4-14.

Serazzi G. (1981), The Dynamic Behaviour of Computer Systems, in Ferrari D. (Ed), Experimental Computer Performance Evaluation, North-Holland, pp 127-163.

Shannon C.E. (1948), A Mathematical theory of communication, Bell System Technical Journal, Vol 27, page 379.

Shemer, J.E. and Heying D.W. (1969), Performance Modelling and Empirical Measurements in a System Designed for Batch and Time-Sharing Users, AFIPS, Vol 35, pp 17-26.

Shemer J.E. and Robertson J.B. (1972), Instrumentation of Time-Shared Systems, IEEE Computer July/August 1972, pp 39-48.

Shen V.Y. et al (1983), Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support, IEEE Transactions on Software Engineering, Vol SE9, No 2, March, pp 155-165.

Sheppard S.B. et al (1979), Modern Coding Practices and Programmer Performance, Computer Vol 12, No 12, December, pp 41-49.

Siegel L.J. et al (1982), Performance Measures for Evaluation Algorithms for SIMD Machines, IEEE Transactions on Software Engineering, Vol SE8, No 4, July, pp 319-331.

Silvester J.A. and Kleinrock L. (1983), On the Capacity of Multihop Slotted ALOHA Networks with Regular Structure, IEEE Transactions on Communications, Vol COM-31, No 8, August, pp 974-982.

Silvester J.A. and Kleinrock L. (1983), On the Capacity of Single-Slotted ALOHA Networks for Various Traffic Matrices and Transmission Strategies, IEEE Transactions on Communications, Vol COM-31, No 8, August, pp 983-991.

Spiegel M.G. (1980), Measuring and Evaluating Performance, Performance Evaluation Review, ACM Sigmetrics, Vol 9, pp 33-34.

Stevens D.F. (1968), System Evaluation of the Control Data 6600, Proc of the IFIP Congress, pp 34-38.

Stucki L.G. (1972), A Prototype Automatic Program Testing Tool, AFIPS, FJCC, pp 829-136.

Sumner (1974), Measurement Techniques in Computer Hardware Design, Infotech Reports, Vol 18, pp 367-390.

Svobodova L. (1973a), Online System Performance Measurements with Software and Hybrid Monitors, Operating Systems Review, Vol 7, No 4, October, pp 45-53.

Svobodova L. (1973b), Measuring Computer System Utilization with a Hardware and a Hybrid Monitor, Performance Evaluation Review, Vol 2, No 4, December.

Svobodova L. (1974), Monitoring and Controlling Performance of a large Computer System, COMPCON Fall, pp 79-82.

Svobodova L. (1976b), Computer System Measurability, IEEE Computer, June, pp 9-17.

Terplan K. (1981), Network Monitor Survey, Computer Performance, Vol 2, No 4, IPC Business Press, pp 158-173.

Terplan K. (1982), Network Performance Reporting, Performance Evaluation Review, ACM Sigmetrics, Vol 11 No 1, pp 156-170.

Tolopka S. (1981), An Event Trace Monitor for the VAX 11/780, Performance Evaluation Review, ACM Sigmetrics, Vol 10, No 3, pp 121-128.

Toong H.D. and Gupta A. (1982), Evaluation Kernels for Microprocessor Performance Analyses, Performance Evaluation, North-Holland, Vol 2, No 1, pp 1-8.

Update on Hardware Monitors (1974), EDP Performance Review, Vol 2, No 10, October, 8 pages.

Von Neumann J. (1946), On the Principles of Large Scale Computing Machines, in Taub A.H. (Ed), Von Neumann's Collected Works, Vol 5, Pergamon, Oxford, pp 1-32.

Warner C.D. (1974), Hardware Monitors: The State of the Art, Infotech Reports, Vol 18, pp 623-631.

Wilner W.T. (1972), Design of the Burroughs B1700, AFIPS, FJCC, Vol 41 pp 489-497.

Wirth N. (1971), Program Development by Stepwise Refinement, Communications of the ACM, Vol 14, No 4, April, pp 221-227.

Witschorik C.A. (1983), The Real-Time Debugging Monitor for the Bell System 1A Processor, Software - Practice and Experience, Vol 13, pp 727-743.

Wulf W.A. and Bell C.G. (1972), C.mmp - A multi-mini-processor, AFIPS, FJCC, Vol 41, Part II, pp 765-777.

Yannacopoulos N.A., Ibbett R.N. and Holgate R.W. (1977), Performance Measurements of the MU5 Primary Instruction Pipeline, Information Processing 77, North-Holland, pp 471-476.

Yuval G. (1975), Gathering Run-Time Statistics Without Black Magic, Software - Practice and Experience, Vol 5, pp 105-108.

13. Appendicies

13.1. Measurement Software

```
PROGRAM TRACE;
(* COLLECT AND ANALYSE EVENT TRACE*)
(*PHILLIP MCKERROW  20.2.84 *)
USES APPLESTUFF;
CONST PLENGTH = 63; OSIZE = 30;
TYPE STRNG = STRING[30];
    LONGINT = INTEGER[12];
    ETRACE = ARRAY [1..64] OF INTEGER;
    PATH = RECORD
        PETIME:LONGINT;
        NOEX:INTEGER;
        MNAME:ARRAY[1..PLENGTH] OF INTEGER;
        MSTIM:ARRAY[1..PLENGTH] OF INTEGER;
        MTIME:ARRAY[1..PLENGTH] OF LONGINT;
        END;
VAR MESG,RUN,TCOMP,SMASK,RTRACE,TALK,LISTEN:STRNG;
   PROBE,STIM,PADD,HTIME,LTIME:ETRACE;
   SLOTNO,I,PHZWD:INTEGER;
   PPOSN,POUT,PIN,PDELM,PCCLN:INTEGER;
   ADD,NUM,ADDR:INTEGER;
   ISTAT,IOUT,PSTATUS : INTEGER;
   (*0-COMMAND MODE, 1-MY TALK ADDRESS, 2-MY LISTEN ADDRESS,
   3-EOI FLAG, 4-SCRN FLAG, 5-PRINTER, 6-SRQ FLAG, 7- SRQ ACK*)
   QUIT,OBJTRAV,LMOD,FOUND:BOOLEAN;
   TERM,DATA,LF,COMMAND,CRET,DOUT,DIN:CHAR;
   TIME:ARRAY[1..64] OF LONGINT;
   STIME,MPERIOD,TTEMP:LONGINT;
   OBJECT:ARRAY[1..OSIZE] OF PATH; (* OBJECT RECORD *)
   TPATH:PATH; (* TEMPORARY PATH RECORD *)
   ONAME:STRNG;
   OBJEX,MSTIME,SMOD,EMOD,J,K:INTEGER;
   (*VARIABLES STARTING WITH P ARE ASSEMBLER GLOBALS*)
   (*AS ARE VARIABLES OF TYPE TRACE*)

(* EXTERNAL ASSEMBLER ROUTINES *)
FUNCTION GETCHAR:CHAR;
EXTERNAL;
PROCEDURE PUTCHAR(DOUT:CHAR);
EXTERNAL;
PROCEDURE IEEEINIT(PSTATUS,POUT:INTEGER);
EXTERNAL;
PROCEDURE RECTRACE;
EXTERNAL;
```

```
PROCEDURE OUTPUT(VAR STR:STRNG;N:INTEGER);
(* SENDS A STRING TO IEEEBUS *)
VAR I,STIM:INTEGER;
   DAT:CHAR;
BEGIN
FOR I := 1 TO N DO
  BEGIN
  DAT := STR[I];
  PUTCHAR(DAT);
  END;
END;

PROCEDURE OBJINIT;
(* INITIALISE OBJECT RECORD*)
BEGIN
WRITELN('ENTER OBJECT NAME');
READLN(ONAME);
WRITELN ('ENTER START AND END MODULES');
READLN(SMOD,EMOD);
MPERIOD := 0;
FOR I:= 1 TO OSIZE DO
  BEGIN
  OBJECT[I].PETIME := 0;
  OBJECT[I].NOEX := 0;
  FOR J := 1 TO PLENGTH DO
    BEGIN
    OBJECT[I].MNAME[J] := 0;
    OBJECT[I].MSTIM[J] := 0;
    OBJECT[I].MTIME[J] := 0;
    END;
  END;
END;

PROCEDURE TIMECONV;
(* CONVERT 32 BIT TIME TO LONG INTEGERS*)
BEGIN
FOR I := 1 TO 64 DO
  BEGIN
  TIME[I] := LTIME[I];
  IF LTIME[I] < 0 THEN TIME[I] := TIME[I] + 65536;
  TTEMP := HTIME[I];
  IF HTIME[I] < 0 THEN TTEMP := TTEMP + 65536;
  TIME[I] := TIME[I] + TTEMP * 65536;
  END;
END;
```

```
PROCEDURE GENOREC;
BEGIN (* GENERATE OBJECT RECORD FROM TRACE BUFFER*)
I := 1;
WHILE I<63 DO
  BEGIN (* INITIALISE PATH RECORD*)
  FOR J := 1 TO PLENGTH DO
    BEGIN
    TPATH.MNAME[J] := 0;
    TPATH.MSTIM[J] := 0;
    TPATH.MTIME[J] := 0;
    END;
  WHILE (PROBE[I] <> SMOD) AND (I < 63) DO I:=I+1;
  J := 1; (* FOUND AN OBJECT *)
  MSTIME := I;
  TPATH.MNAME[J] := PROBE[I];
  OBJTRAV := FALSE;
  LMOD := FALSE;
  WHILE (OBJTRAV = FALSE) AND (J < PLENGTH) AND (I < 63) DO
    BEGIN (* TRAVERSE OBJECT ONE MODULE AT A TIME *)
    I := I+1; (*GO TO NEXT TUPLE*)
    IF PROBE[I] = PROBE[I-1] THEN TPATH.MSTIM[J] := STIM[I] (*STIMULUS*)
    ELSE
      BEGIN (* EVENT TUPLE *)
        IF SMOD = EMOD THEN
          BEGIN (* TRACING ONE MODULE*)
          I := I - 1;
          LMOD := TRUE;
          END;
      IF LMOD THEN
        BEGIN (* END OF OBJECT *)
        OBJTRAV := TRUE;
        TPATH.MTIME[J] := TIME[I+1] - TIME[MSTIME];
        TPATH.PETIME := 0; (* PATH EXECUTION TIME *)
        FOR K := 1 TO J DO TPATH.PETIME := TPATH.PETIME + TPATH.MTIME[K];
        FOUND := FALSE;
        K := 1;
        WHILE (OBJECT[K].PETIME <> 0) AND (NOT FOUND) AND (K < OSIZE) DO
          BEGIN (* LOOK FOR A KNOWN PATH *)
          IF TPATH.PETIME = OBJECT[K].PETIME THEN
            BEGIN (*FOUND A KNOWN PATH*)
            OBJECT[K].NOEX := OBJECT[K].NOEX + 1;
            FOUND := TRUE;
            END
          ELSE K := K + 1;
          END;
```

```
        IF NOT FOUND THEN
          BEGIN (* NEW PATH, COPY PATH RECORD *)
          OBJECT[K] := TPATH;
          OBJECT[K].NOEX := 1;
          END;
          IF SMOD = EMOD THEN I := I + 1;
        END
      ELSE (* END OF MODULE*)
        BEGIN
        J := J + 1;
        TPATH.MTIME[J-1] := TIME[I] - TIME[MSTIME];
        MSTIME := I;
        TPATH.MNAME[J] := PROBE[I];
        IF PROBE[I] = EMOD THEN
          BEGIN (*LAST MODULE IN OBJECT*)
          LMOD := TRUE;
          I := I - 1; (*STAY AT THIS TUPLE*)
          END;
        END;
      END;
    END;
  END;
MPERIOD := MPERIOD + TIME[64] - TIME[1];
END;

PROCEDURE PRNTPATH(K:INTEGER);
BEGIN   (*PROCEDURE TO PRINT A PATH RECORD*)
WRITELN('PATH ',K,' TIME ',OBJECT[K].PETIME,' NOEX ',OBJECT[K].NOEX);
J := 1;
WHILE (J<PLENGTH) AND (OBJECT[K].MNAME[J] <> 0) DO
BEGIN
WRITE('MOD ');
FOR I:=J TO J+8 DO WRITE(OBJECT[K].MNAME[I]:8);
WRITELN;
WRITE('TIME');
FOR I:=J TO J+8 DO WRITE(OBJECT[K].MTIME[I]:8);
WRITELN;
WRITE('STIM');
FOR I:=J TO J+8 DO WRITE(OBJECT[K].MSTIM[I]:8);
WRITELN;
J := J+9;
END;
WRITELN;
WRITELN;
END;
```

```
PROCEDURE ANALDAT;
BEGIN   (*ANALYSE OBJECT RECORD AND PRODUCE CALCULATED VALUES*)
WRITELN ('DATA ANALYSIS FOR OBJECT ',ONAME);
WRITELN;
WRITE('PATH ');
FOR K := 1 TO OSIZE DO WRITE(K:8); (* PATH NUMBER *)
WRITELN;
WRITE('TIME ');
FOR K := 1 TO OSIZE DO WRITE(OBJECT[K].PETIME:8); (* PATH EXECUTION TIME *)
WRITELN;
WRITE('NOEX ');
OBJEX := 0;
FOR K:= 1 TO OSIZE DO
BEGIN
  OBJEX := OBJEX + OBJECT[K].NOEX;
  WRITE(OBJECT[K].NOEX:8); (* PATH EXECUTION COUNT *)
END;
WRITELN;
WRITE('FREQ ');
IF OBJEX>0 THEN (* PATH EXECUTION FREQUENCY *)
  FOR K:= 1 TO OSIZE DO WRITE(OBJECT[K].NOEX * 100.0 / OBJEX:8:2);
WRITELN;
WRITE('UTLZ ');
IF MPERIOD>0 THEN
  FOR K:=1 TO OSIZE DO (* PATH UTILIZATION *)
    WRITE(OBJECT[K].NOEX * OBJECT[K].PETIME * 10000 DIV MPERIOD:8);
WRITELN;
WRITELN;
IF MPERIOD>0 THEN (* OBJECT THROUGHPUT *)
  WRITELN('PERIOD ',MPERIOD,' THROUGHPUT ',OBJEX * 10000000 DIV MPERIOD);
WRITELN;
WRITELN;
END;

PROCEDURE DISPREC;
(*DISPLAY THE OBJECT RECORD*)
BEGIN
WRITELN('OBJECT ',ONAME,' MEAS PERIOD ',MPERIOD);
K := 1;
WHILE (OBJECT[K].PETIME <> 0) AND (K < OSIZE) DO
  BEGIN
  PRNTPATH(K);
  K := K + 1;
  END;
END;
```

```
PROCEDURE INITDATA;
BEGIN (*SETS UP VARIABLES AND MESSAGES*)
QUIT := FALSE;
PHZWD := 40;
PPOSN := 0;
PCCLN := 0;
PDELM := 32;
CRET := CHR(13);
LF := CHR(10);
TERM := CHR(141);
TALK := '@?A/ '
LISTEN := '@?O! '
RUN := 'RU ';
TCOMP := 'SB1* ';
SMASK := 'SM,0,1,0,0,0,0,0,0 ';
RTRACE := 'LR3* ';
RTRACE[5] := LF;
RUN[3] := LF;
TCOMP[5] := LF;
SMASK[19] := LF;
TALK[5] := CRET;
LISTEN[5] := CRET;
ISTAT := 12288;  (* IEEE CARD IN SLOT 3 *)
IOUT := -15616;
END;

PROCEDURE RUNAN;
BEGIN (* RUN ANALYSER TO COLLECT DATA*)
PSTATUS := ISTAT;
POUT := IOUT;
IEEEINIT(PSTATUS,POUT);
OUTPUT(TALK,5);
OUTPUT(RUN,3);
END;

PROCEDURE COLDAT;
BEGIN (* READ AND PROCESS DATA*)
IF COMMAND = 'M' THEN RUNAN;
REPEAT (*LOOK FOR TRACE COMPLETE*)
  OUTPUT(TCOMP,5);
  OUTPUT(LISTEN,5);
  DIN := GETCHAR;
  PSTATUS := ISTAT;
  POUT := IOUT;
  IEEEINIT(PSTATUS,POUT);
  OUTPUT(TALK,5);
UNTIL ORD(DIN) = 128;
OUTPUT(RTRACE,5);
OUTPUT(LISTEN,5);
RECTRACE; (* RECORD TRACE IN ARRAYS PROBE,STIM,PADD,HTIME,LTIME *)
IF COMMAND = 'C' THEN RUNAN; (*START NEXT TRACE WHILE PROCESSING DATA*)
TIMECONV; (* ONVERT TIME VALUES FROM INTS TO LONGS*)
GENOREC; (*ADD TRACE TO OBJECT RECORD*)
END;
```

```
BEGIN            (* MAIN PROGRAM*)
(*INITIALISATION*)
WRITELN('EVENT TRACE RECORDING AND ANALYSIS');
WRITELN('ANALYSER SETUP: POD4 - MODULE IDENTIFIER, POD3 - STIMULUS');
WRITELN('POD2,1 - ADDRESS, CLOCK SLOPE -VE, COUNT TIME');
INITDATA;
OBJINIT;
PSTATUS := ISTAT;
POUT := IOUT;
IEEEINIT(PSTATUS,POUT);
OUTPUT(TALK,5);
OUTPUT(SMASK,19);
REPEAT    (* MENU*)
  WRITELN('      DATA COLLECTION');
  WRITELN('ENTER R ECORD,C ONT MEAS,D ISPLAY,M EASURE,I NIT,Q UIT');
  READLN(COMMAND);
  CASE COMMAND OF
    'Q' : QUIT := TRUE;
    'R' : DISPREC; (* DISPLAY OBJECT RECORD*)
    'M' : COLDAT; (* COLLECT DATA*)
    'I' : OBJINIT; (* INITIALISE OBJECT RECORD*)
    'C' : BEGIN   (* CONTINUOUS MONITORING*)
          WRITELN('CONTINUOUS MONITORING');
            RUNAN; (* INITIAL TRACE*)
          REPEAT
            COLDAT; (* RUN ANALYSER TO COLLECT DATA*)
          UNTIL KEYPRESS;
          READ(COMMAND);
          END;
    'D' : BEGIN   (*DATA DISPLAY*)
          REPEAT
           WRITELN('      DATA ANALYSIS');
           WRITELN('ENTER R ECORD,P ATH,A NALYSE,Q UIT');
           READLN(COMMAND);
           CASE COMMAND OF
             'Q' : QUIT := TRUE;
             'P' : BEGIN (* PRINT A PATH*)
                   WRITE('ENTER PATH NUMBER ');
                   READLN(K);
                   IF (K>0) AND (K<OSIZE) THEN PRNTPATH(K)
                     ELSE WRITELN('INVALID PATH');
                   END;
             'R' : DISPREC; (* DISPLAY OBJECT RECORD*)
             'A' : ANALDAT; (* CALCULATE VALUES*)
           END;
          UNTIL QUIT;
          QUIT := FALSE;
          END;
  END;
UNTIL QUIT;
WRITELN ('BYE BYE');
END.
```

```
;ROUTINES FOR TRACE TRANSFER FROM ANALYSER TO APPLE
;THE UNIVERSITY OF WOLLONGONG
;PHILLIP MCKERROW  20.2.84
;
;MACRO DEFINITIONS
;
; MACRO POP ADDRESS
;POPS 16 BIT ARG FROM STACK TO ASSRESS
     .MACRO POP
     PLA        ;PULL LS BYTE
     STA %1
     PLA        ;PULL MS BYTE
     STA %1+1
     .ENDM
;
; MACRO PUSH ADDRESS
;PUSHES 16 BIT ARG TO STACK
     .MACRO PUSH
     LDA %1+1
     PHA        ;PUT MS BYTE
     LDA %1
     PHA        ;PUT LS BYTE
     .ENDM

; MACRO MOVE 2 BYTES FROM %1 TO %2
     .MACRO MOVE
     LDA %1     ;MOVE LS BYTE
     STA %2
     LDA %1+1   ;MOVE MS BYTE
     STA %2+1
     .ENDM

; MACRO MOVEB BYTE FROM %1 TO %2
     .MACRO MOVEB
     LDA %1
     STA %2
     .ENDM
;
```

```
     .FUNC GETCHAR
;FUNCTION GETCHAR:CHAR;
; READ A CHARACTER FROM IEEE BUS
     .REF INBYT
RETURN .EQU 0
     POP RETURN
     PLA     ;REMOVE STACK BIAS
     PLA
     PLA
     PLA
     JSR INBYT ;CALL CHARACTER INPUT ROUTINE IN RECTRACE
     TAY
     LDA #00
     PHA
     TYA
     PHA     ;RETURN CHARACTER
     PUSH RETURN
     RTS

     .PROC IEEEINIT,2
;PROCEDURE IEEEINIT(PSTATUS,POUT:INTEGER);
;INITIALISE IEEEBUS INTERFACE
RETURN .EQU 0
STAT   .EQU 7
SLOT   .EQU 8
OUT    .EQU 036
ICSR   .EQU 0C081
     POP RETURN
     POP OUT
     POP STAT
      ;SELECT CARD
     BIT 0CFFF   ; DESLECT PERIPHERALS
     BIT 0CBFF   ; DESELECT IEEECARD
     LDX #00
     LDA @OUT,X    ;SELECT IEEE CARD
; INITIALISE INTERFACE
     LDX SLOT
     LDA #2F     ;REN,ATN,IFC = 0
     STA ICSR,X   ;IEEE CSR
     LDA #3F     ;RELEASE IFC
     STA ICSR,X
     PUSH RETURN
     RTS
```

```
      .PROC PUTCHAR,1
; PROCEDURE PUTCHAR(DOUT:CHAR);
; WRITE A CHARACTER TO IEEE BUS
      .PUBLIC PSTATUS,PHZWD,PPOSN,POUT
      .PUBLIC PIN,PDELM,PCCLN
RETURN .EQU 0
DATA   .EQU 2
STAT   .EQU 7
HZWD   .EQU 21
POSN   .EQU 24
OPHK   .EQU 36
INHK   .EQU 38
XREG   .EQU 46
DELM   .EQU 5FB
CCLN   .EQU 7FB
ADDR   .EQU 0C0D0
OUTPUT .EQU 0C32B
      POP RETURN
      POP DATA
      LDA #0F0
      STA ADDR ; SAVE/RESTORE VARIABLES
      PUSH OPHK
      PUSH INHK
      PUSH XREG
       PUSH DELM
       PUSH CCLN
      MOVE PSTATUS,STAT
      MOVEB PHZWD,HZWD
      MOVEB PPOSN,POSN
      MOVE  POUT,OPHK
      MOVE  PIN,INHK
      MOVEB PDELM,DELM
      MOVEB PCCLN,CCLN
      LDA #0F1
      STA ADDR
      LDA   DATA
      JSR OUTPUT ;CALL FIRMWARE ROUTINE
      LDA #0F0
      STA ADDR ;SAVE/RESTORE VARIABLES
      MOVE STAT,PSTATUS
      MOVEB HZWD,PHZWD
      MOVEB POSN,PPOSN
      MOVEB DELM,PDELM
      MOVEB CCLN,PCCLN
      POP CCLN
      POP DELM
      POP XREG
      POP INHK
      POP OPHK
      LDA #08
      STA ADDR
      PUSH RETURN
      RTS
```

```
     .PROC RECTRACE
; PROCEDURE RECTRACE;
;READS A TRACE FROM THE ANALYSER SCREEN
;AND SAVES IT IN ARRAYS
;FORMAT: POD4 - MODULE IDENTIFIER, POD3 - STIMULUS
;POD2 - ADDRESS HIGH, POD1 - ADDRESS LOW
     .PUBLIC PSTATUS,PROBE,STIM,PADD
     .PUBLIC HTIME,LTIME
     .DEF INBYT
ICSR  .EQU 0C081
ISRG  .EQU 0C082
IDIN  .EQU 0C083
     LDY #0B
SKIP  JSR INBYT
     DEY
     BPL SKIP
     LDY #00   ;WANT TO READ 128 BYTES
EVENT JSR INBYT
     STA PROBE,Y
     LDA #00
     STA PROBE+1,Y
     JSR INBYT
     STA STIM,Y
     LDA #00
     STA STIM+1,Y
     JSR INBYT
     STA PADD+1,Y
     JSR INBYT
     STA PADD,Y
     INY
     INY
     BPL EVENT  ;GOES NEGATIVE ONBYTE 128
     LDY #00
TIME  JSR INBYT
     STA HTIME+1,Y
     JSR INBYT
     STA HTIME,Y
     JSR INBYT
     STA LTIME+1,Y
     JSR INBYT
     STA LTIME,Y
     INY
     INY
     BPL TIME
CRC   JSR INBYT
     LDA PSTATUS
     AND #08
     BEQ CRC
     JSR INBYT   ;RESET EOI
     RTS
```

```
INBYT  LDX  PSTATUS+1 ;SUBROUTINE TO INPUT A CHAR FROM IEEE BUS
       LDA  PSTATUS
       AND  #08
       BNE  EOI      ;EOI FLAG IS SET
       LDA  #77
       STA  ICSR,X   ;NRFD SET TO 1
DAV    LDA  ISRG,X
       AND  #02      ;DAV = 0
       BNE  DAV
       LDA  ISRG,X
       AND  #01      ;EOI = 1
       BNE  NEND
       LDA  PSTATUS
       ORA  #08      ;SET DATA END FLAG
       STA  PSTATUS
NEND   LDA  IDIN,X   ;READ DATA REG
       EOR  #0FF     ;INVERT BYTE
       PHA
       LDA  #73      ;SET NDAC,NRFD TO 0
       STA  ICSR,X
       LDA  #7B      ;SET NDAC = 1
       STA  ICSR,X
NDAV   LDA  ISRG,X
       AND  #02      ;DAV = 1
       BEQ  NDAV
NDAC   LDA  #73      ;SET NDAC,NRFD TO 0
       STA  ICSR,X
       PLA
       JMP  END
EOI    LDA  PSTATUS
       AND  #0F7     ;CLR MLA,EOI
       STA  PSTATUS
       LDA  #0D
END    RTS
       .END
```

13.2. Communications Test Software

```
PROGRAM INTIEEE;
(*PROGRAM TO TEST IEEE 488 INTERFACE*)
(*PHILLIP MCKERROW   16.2.84 *)
CONST SIZE = 30;
TYPE STRNG = STRING[SIZE];
    PA = PACKED ARRAY[0..1] OF 0..255;
    MAGIC = RECORD
          CASE BOOLEAN OF
          TRUE : (INT:INTEGER);
          FALSE : (PTR:^PA);
          END;
VAR CHEAT:MAGIC;
   TALK,LISTEN,CMD,MESG:STRNG;
   SLOTNO,I,PHZWD:INTEGER;
   PPOSN,POUT,PIN,PDELM,PCCLN:INTEGER;
   ADD,NUM,ADDR:INTEGER;
   PSTATUS : PACKED ARRAY[0..15] OF BOOLEAN;
   (*0-COMMAND MODE, 1-MY TALK ADDRESS, 2-MY LISTEN ADDRESS,
   3-EOI FLAG, 4-SCRN FLAG, 5-PRINTER, 6-SRQ FLAG, 7- SRQ ACK*)
   QUIT:BOOLEAN;
   TERM,DATA,LF,COMMAND,CRET,DOUT,DIN:CHAR;
(*VARIABLES STARTING WITH P ARE ASSEMBLER GLOBALS*)
PROCEDURE GETCHAR;
EXTERNAL;
PROCEDURE PUTCHAR(DOUT:CHAR);
EXTERNAL;
PROCEDURE IEEEINIT(SLOT:INTEGER);
EXTERNAL;
PROCEDURE POKE(DATA,ADDR:INTEGER);
EXTERNAL;
FUNCTION PEEK(ADDR:INTEGER):INTEGER;
EXTERNAL;
PROCEDURE TEST;
EXTERNAL;
PROCEDURE OUTPUT(VAR STR:STRNG;N:INTEGER);
(* SENDS A STRING TO IEEEBUS *)
VAR I,STIM:INTEGER;
   DAT:CHAR;
BEGIN
POKE(8,ADDR);
FOR I := 1 TO N DO
  BEGIN
  DAT := STR[I];
  STIM := ORD(DAT);
  POKE(STIM,ADDR+2);
  PUTCHAR(DAT);
  END;
POKE(7,ADDR);
END;
```

```
BEGIN
WRITELN ('INTERACTIVE COMMUNICATION TO ANALYSER');
ADDR := -16176;  (* PROBE IS IN SLOT 5 *)
SLOTNO := 3;  (* IEEE CARD IN SLOT 3 *)
POKE(1,ADDR);
IEEEINIT(SLOTNO);
POKE(2,ADDR);
QUIT := FALSE;
PHZWD := 40;
PPOSN := 0;
PCCLN := 0;
PDELM := 32;
CRET := CHR(13);
LF := CHR(10);
TERM := CHR(141);
TALK := '@?A/ '
LISTEN := '@?O! '
TALK[5] := CRET;
LISTEN[5] := CRET;
MESG := '                    ';
REPEAT
  POKE(7,ADDR);
  WRITELN('TALK LISTEN OUT IN RESET QUIT EXMEM
        MODMEM PROBETIME SINGLEBYTE');
  READLN(COMMAND);
  CASE COMMAND OF
   'T' : BEGIN (* APPLE TO TALK*)
        POKE(3,ADDR);
        OUTPUT(TALK,5);
        END;
   'L' : BEGIN (* APPLE TO LISTEN *)
        POKE(4,ADDR);
        OUTPUT(LISTEN,5);
        END;
   'E' : BEGIN
        WRITE('ADDRESS PLEASE ');
        READLN(ADD);
        NUM := PEEK(ADD);
        WRITELN(' ',NUM);
        END;
   'M' : BEGIN
        WRITE('DATA,ADDRESS PLEASE ');
        READLN(NUM,ADD);
        POKE(NUM,ADD);
        END;
```

```
  'O' : BEGIN (* OUTPUT STRING *)
        POKE(5,ADDR);
        I := 0;
        WRITELN('ENTER DATA 1..',SIZE);
        WHILE (NOT EOLN(INPUT)) AND (I<SIZE) DO
          BEGIN
          I := I+1;
          READ (DATA);
          MESG[I] := DATA;
          IF I>= SIZE THEN WRITE('BUFFER FULL');
          END;
        READLN;
        MESG[I] := LF;
        OUTPUT(MESG,I);
        END;
  'R' : BEGIN (* RESET INTERFACE*)
        IEEEINIT(SLOTNO);
        END;
  'I' : BEGIN (* INPUT STRING *)
        POKE(6,ADDR);
        REPEAT
          GETCHAR;
          I := ORD(DIN);
          POKE(I,ADDR+2);
          WRITE(DIN);
        UNTIL PSTATUS[3] = TRUE;
        WHILE PSTATUS[3] DO
          BEGIN
          GETCHAR;
          WRITELN(DIN);
          END;
        END;
  'P' : BEGIN (* PROBE TIMING*)
        CHEAT.INT := ADDR;
        POKE(128,ADDR);
        POKE(129,ADDR);
        POKE(130,ADDR);
        CHEAT.PTR^[0] := 131;
        CHEAT.PTR^[0] := 132;
        CHEAT.PTR^[0] := 133;
        TEST; (* ASSEMBLER WRITE*)
        END;
  'S' : BEGIN (* READ ONE BYTE*)
        GETCHAR;
        WRITELN(DIN,ORD(DIN));
        END;
  'Q' : QUIT := TRUE;
  END;
UNTIL QUIT;
WRITELN ('BYE BYE');
POKE(0,ADDR);
END.
```

```
;ROUTINES FOR IEEECARD INPUT/OUTPUT
;THE UNIVERSITY OF WOLLONGONG
;PHILLIP MCKERROW   16.2.84
;
;
;MACRO DEFINITIONS
;
;  MACRO POP ADDRESS
;POPS 16 BIT ARG FROM STACK O ASSRESS
;
;
      .MACRO POP
      PLA         ;PULL LS BYTE
      STA %1
      PLA         ;PULL MS BYTE
      STA %1+1
      .ENDM
;
;  MACRO PUSH ADDRESS
;PUSHES 16 BIT ARG TO STACK
;
      .MACRO PUSH
      LDA %1+1
      PHA         ;PUT MS BYTE
      LDA %1
      PHA         ;PUT LS BYTE
      .ENDM
;  MACRO MOVE DATA FROM %1 TO %2
;MOVES 2 BYTES
;
      .MACRO MOVE
      LDA %1      ;MOVE LS BYTE
      STA %2
      LDA %1+1    ;MOVE MS BYTE
      STA %2+1
      .ENDM
;  MACRO MOVEB BYTE FROM %1 TO %2
;
      .MACRO MOVEB
      LDA %1
      STA %2
      .ENDM
```

```
      .PROC PUTCHAR,1
; PROCEDURE PUTCHAR(DOUT:CHAR);
; WRITE A CHARACTER TO IEEE BUS
      .PUBLIC PSTATUS,PHZWD,PPOSN,POUT
      .PUBLIC PIN,PDELM,PCCLN
RETURN .EQU 0
DATA   .EQU 2
STAT   .EQU 7
HZWD   .EQU 21
POSN   .EQU 24
OPHK   .EQU 36
INHK   .EQU 38
XREG   .EQU 46
DELM   .EQU 5FB
CCLN   .EQU 7FB
ADDR   .EQU 0C0D0
OUTPUT .EQU 0C32B
      POP RETURN
      POP DATA
      LDA #0F0
      STA ADDR
      PUSH OPHK
      PUSH INHK
      PUSH XREG
       PUSH DELM
       PUSH CCLN
      MOVE PSTATUS,STAT
      MOVEB PHZWD,HZWD
      MOVEB PPOSN,POSN
      MOVE  POUT,OPHK
      MOVE  PIN,INHK
      MOVEB PDELM,DELM
      MOVEB PCCLN,CCLN
      LDA #0F1
      STA ADDR
      LDA   DATA
      JSR OUTPUT
      LDA #0F0
      STA ADDR
      MOVE STAT,PSTATUS
      MOVEB HZWD,PHZWD
      MOVEB POSN,PPOSN
      MOVEB DELM,PDELM
      MOVEB CCLN,PCCLN
      POP CCLN
      POP DELM
      POP XREG
      POP INHK
      POP OPHK
      LDA #08
      STA ADDR
      PUSH RETURN
      RTS
```

```
     .PROC GETCHAR
; PROCEDURE GETCHAR;
; READ A CHARACTER FROM IEEE BUS
     .PUBLIC PSTATUS,DIN
RETURN .EQU 0
STAT   .EQU 7
SLOT   .EQU 8
ICSR   .EQU 0C081
ISRG   .EQU 0C082
IDIN   .EQU 0C083
      POP RETURN
      MOVE PSTATUS,STAT
      LDX  SLOT
      LDA  STAT
      AND  #08
      BNE  CROT
      LDA  #77
      STA  ICSR,X
 DAV    LDA  ISRG,X
      AND  #02
      BNE  DAV
      LDA  ISRG,X
      AND  #01
      BNE  NEND
      LDA  STAT
      ORA  #08
      STA  STAT
 NEND   LDA  IDIN,X
      EOR  #0FF
      STA  DIN
      LDA  #73
      STA  ICSR,X
      LDA  #7B
      STA  ICSR,X
 NDAV   LDA  ISRG,X
      AND  #02
      BEQ  NDAV
 NDAC   LDA  #73
      STA  ICSR,X
      JMP  END
 CROT   LDA  STAT
      AND  #0F7
      STA  STAT
      LDA  #0D
      STA   DIN
 END    LDA   #0
      STA   DIN+1
      MOVE STAT,PSTATUS
      PUSH RETURN
      RTS
```

```
      .PROC IEEEINIT,1
;PROCEDURE IEEEINIT(SLOT:INTEGER);
;INITIALISE IEEEBUS INTERFACE
      .PUBLIC PSTATUS
      .PUBLIC POUT
RETURN .EQU 0
STAT   .EQU 7
SLOT   .EQU 8
OUT    .EQU 036
ICSR   .EQU 0C081
      POP RETURN
      POP STAT
      PUSH OUT
; GET CARD ROM ADDRESS
      LDA STAT
      STA SLOT   ;POP REVERSES THEM
      AND #0F
      ORA #0C0    ;ADD BASE
      STA OUT+1   ;ROM ADDRESS
      LDA #0
      STA OUT
; INITIALISE STATUS AND SELECT CARD
      STA STAT
      BIT 0CFFF   ; DESLECT PERIPHERALS
      BIT 0CBFF   ; DESELECT IEEECARD
      TAX
      LDA @OUT,X    ;SELECT IEEE CARD
      LDA SLOT
      AND #0F
      ASL A       ;CHANGE 0N TO N0
      ASL A
      ASL A
      ASL A
      STA SLOT    ;SAVE
; INITIALISE INTERFACE
      TAX
      LDA #2F     ;REN,ATN,IFC =0
      STA ICSR,X  ;IEEE CSR
      LDA #3F     ;RELEASE IFC
      STA ICSR,X
; SAVE GLOBAL INFORMATION
      MOVE STAT,PSTATUS
      MOVE OUT,POUT
      POP OUT
      PUSH RETURN
      RTS
;
```

```
     .PROC TEST
;PROCEDURE TO TEST PROBE TIMING
ADDR   .EQU  0C0D0
     LDA   #0E0
     STA   ADDR
     LDA   #0E1
     STA   ADDR
     LDA   #0E2
     STA   ADDR
     RTS

     .FUNC PEEK,1
; FUNCTION PEEK(ADDRESS:INTEGER):INTEGER;
;RETURNS CONTENTS OF SPECIFIED ADDRESS
;8 BITS OF DATA RETURNED IN LS BYTE
;MS BYTE SET TO ZERO
;
RETURN .EQU 0     ;STORAGE FOR RETURN ADDRESS
ADDR   .EQU 2     ;ADDRESS OF DATA
     POP RETURN       ;GET RETURN ADDRESS
     PLA              ;STACK BIAS
     PLA
     PLA
     PLA
     POP ADDR         ;GET ADDRESS
     LDA #0
     PHA              ;SET MS BYTE TO 0
     LDY #0
     LDA @ADDR,Y      ;GET DATA
     PHA              ;RETURN DATA
     PUSH RETURN
     RTS              ;GO BACK
;
     .PROC POKE,2
; PROCEDURE POKE(DATA,ADDR:INTEGER)
;PROCEDURE TO WRITE TO ADDRESS
;
RETURN .EQU 0
ADDR   .EQU 2
     POP RETURN   ;SAVE RETURN ADDRESS
     POP ADDR     ;MEMORY LOCATION
     LDX #0
     PLA          ;GET OUTPUT
     STA @ADDR,X  ;POKE
     PLA          ;CLEAN UP STACK
     PUSH RETURN
     RTS          ;GO BACK
     .END
```